بسمه‌تعالی

عنوان مستند:

برنامه‌نویسی امن جاوا

مرکز ماهر

تابستان ۱۳۹۶

فهرست مطالب

فهرست جداول و اشکال

فصل اول: مقدمه‌ای بر امنیت جاوا

به‌طور کلی، مدل امن جاوا، یک مسیر قابل اعتماد برای ساخت برنامه‌های کاربردی، توزیع‌شده و امن جاوا است. پیاده‌سازی مدل امن جاوا، بستری از جزئیات و ابزارها را، که برای هر معماری جاوا ضروری است، نشان می دهد. در این فصل، مروری بر تاریخچه‌ی امنیت جاوا ارائه نموده‌ایم. در این فصل، نکات زیر بررسی خواهند شد:

- تاریخچه‌ی امنیت در جاوا
- مروری بر مدل امنیت
- نقش تشخیص‌گر بایت کد ماشین مجازی جاوا (JVM)[1] در تأمین امنیت
- جنبه‌های امنیت بارکننده‌ی کلاس JVM
- جنبه‌های امنیت مدیر امنیت JVM
- مبنای رمزنگاری زیرساخت که توسط پلت‌فرم جاوا فراهم می‌شود

۱.۱ تاریخچه‌ی امنیت در جاوا

جاوا در اصل جهت استفاده‌ی توسعه‌دهندگان، به‌منظور ساخت آپلت‌های[2] جاوا پیشنهاد شد. آپلت‌های جاوا، قابلیت دانلود مستقیم در یک مرورگر وب را برای کد فراهم می‌کنند. این تکنولوژی، از اولین فناوری‌های وارد کردن مرورگرهای وب به قالبی که می‌تواند از اجرای برنامه‌های کاربردی روی وب پشتیبانی کند، به‌شمار می‌رفت. چنین چارچوبی، که وعده‌ی ارائه‌ی یک الگو جدید را برای محاسبات داده‌است، کاملا متفاوت با رایانه‌های رومیزی سنتی است. با محاسبات رومیزی، برنامه‌های کاربردی روی ماشین بارگذاری و اجرا می‌شوند. اگرچه، به به‌روزرسانی‌هایی در نرم‌افزار کاربردی نیاز است تا توزیعاتی از منابعی چون CDها و دیسکت‌ها را در اختیار داشته باشید. سپس، به‌روزرسانی‌های خود را بارگذاری می‌کنید. آپلت‌های جاوا، الگو جدیدی ارائه دادند که کد موبایل، به‌صورت خودکار روی مرورگر وب دانلود شود و هنگامی‌که یک وب‌سایت را مجدداً مشاهده می‌کنید، به‌صورت خودکار، به‌روزرسانی گردد.

---

[1] Java Virtual Machine (JVM)

[2] Applets

اگرچه، کارایی شبکه کمک شایانی در رسیدن به این چشم‌انداز بزرگ نموده‌است، اما محدود کردن اندازه‌ی آپلِت‌های جاوا، دلیلی منطقی است که کاربران برای دانلود و در نتیجه محدودیت پیچیدگی‌های دانلود شدن برنامه‌های کاربردی، درنظر دارند. علاوه برآنها، کارایی JVM مجهز به مرورگرهای وب نیز، مانع از تکثیر آپلِت‌های جاوا در اینترنت شده‌است.

مدل امنیتی نسخه‌ی 1.0 جاوا، تا حد زیادی محدودکننده بود. تصور فراهم کردن برنامه‌های کاربردی قابل دانلود از طریق وب، فرونشست، زیرا برنامه‌های کاربردی نمی‌توانند عملیات کلیدی، نظیر دست‌یابی به فایل یا ایجاد اتصالات جدید در شبکه، را تشخیص دهند. اگر فروشندگان، با مرورگر وب از طریق کدهای دور، مانند کدهای محلی، رفتار کنند، مسیر برای کدهای مخرب باز خواهد بود. از این‌رو، مدل همه یا هیچ در نسخه‌ی 1.1 جاوا جایگزین شد.

با استفاده از مدل کد قابل اعتماد، می‌توانید به‌صورت سفارشی طراحی نمایید که آیا کد امضاشده توسط یک تولیدکننده، معتبر است یا خیر و آیا می‌تواند به تمامی منابع اجازه دسترسی داشته‌باشد. بنابراین، ممکن است به تعدادی از کدهای جاوا از مایکروسافت اعتماد کنید تا این قابلیت را داشته باشند که با دسترسی کامل به منابع سیستم، در کنار مرورگر شما اجرا شوند. هنگامی‌که یکی از تولیدات مایکروسافت را روی سیستم خود نصب می‌نمایید، تا حد زیادی به مایکروسافت اعتماد می‌کنید. شرکتی همانند مایکروسافت باید کد یا برنامه‌های خود را امضا کند تا شما بتوانید به حقیقت تشخیص دهید که این کد، متعلق به این شرکت است. بنابراین، آپلِت‌های امضاشده‌ی جاوا، به‌صورت تضمینی، به تمامی منابع سیستم، دسترسی خواهند داشت. این در حالی است که کدهای غیرقابل اعتماد، به یک جعبه‌شنی [۳] محدود می‌شوند.

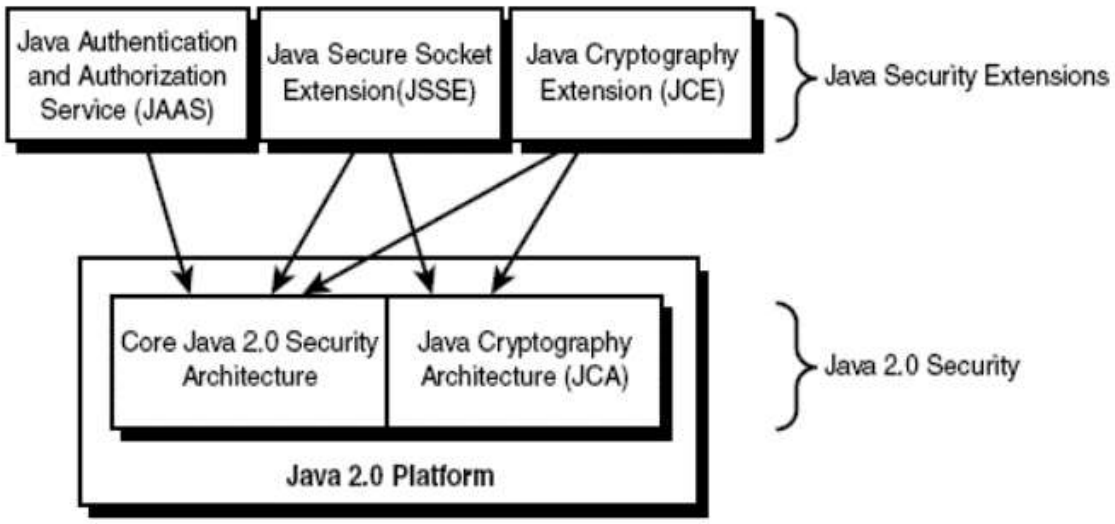
پلت‌فرم نسخه‌ی 2 جاوا (همچنین جاوا 1.2 نامیده می‌شود)، توسط مدل امن دقیق‌تری، امنیت برنامه‌های کاربردی را تکمیل کرد. اکنون، کدهای محلی و از راه دور مشابه، می‌توانند منطبق بر سیاست پیکربندی، به دامنه‌های خاصی از منابع، محدود شوند. دامنه‌های در دسترس و پیکربندی سیاست‌های امنیتی، جاوا

[۳] Sandbox

2.0 را بیش از پیش قابل انعطاف کرد. بنابراین، نگرانی توسعه‌دهندگان و تمرکز آنها بر امنیت دستگاه‌ها، مبنی بر تمایز بین کدهای دور و کدهای محلی، از بین رفت.

۲,۱ معماری امنیتی جاوا

شکل ۱-۱، جزئیات اصلی مجموعه استانداردهای APIها را به تصویر می‌کشد و از مکانیزم‌هایی استفاده می‌شود تا امنیت برنامه‌های کاربردی مبتنی بر جاوا 2 را فراهم کنند. در نیمه‌ی پایینی نمودار، هسته‌ی معماری امنیتی جاوا 2 و معماری رمز جاوا (JCA)[۴] وجود دارند. پلت‌فرم امنیت جاوا 2، حاصل ترکیب این دو است. طبق نیمه‌ی بالایی نمودار، استاندارد گسترده‌ی امن جاوا، جدا از پلت‌فرم جاوا 2 است، اما هنوز به جنبه‌های مختلفی از آن وابستگی دارد.



شکل ۱-۱: مؤلفه‌های استاندارد معماری امنیت جاوا

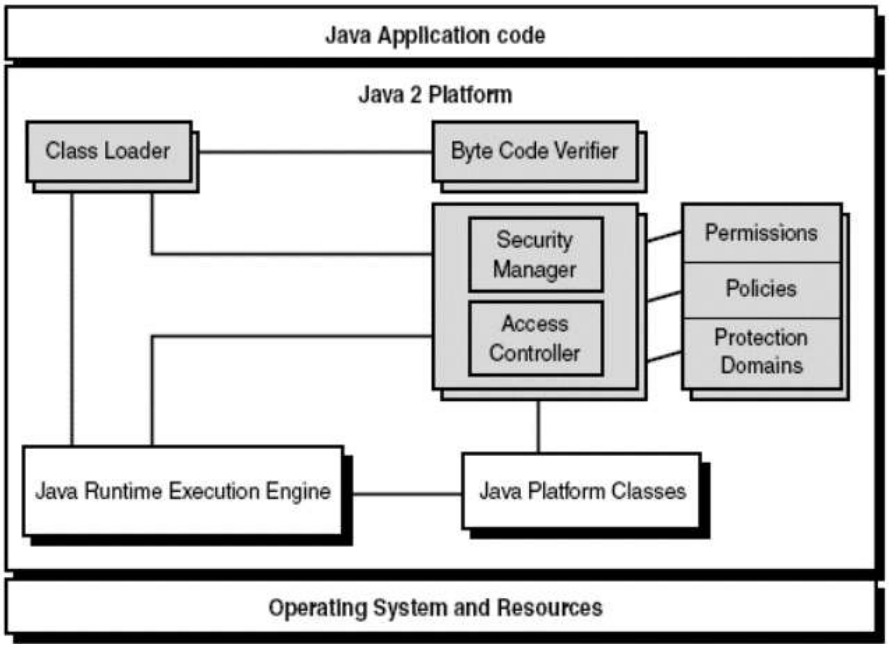
۳,۱ معماری امنیتی هسته‌ی جاوا 2.0

شکل ۱-۲، معماری امنیتی هسته‌ی جاوا 2.0 را نشان می‌دهد. سیستم‌عامل، منابع، و کد اجرایی جاوا، روی پلت‌فرم جاوا 2 قرار دارند. تکه‌های این معماری، که در قالب هسته‌ی امنیت جاوا هستند، عبارتند از:

[۴] Java Cryptography Architecture (JCA)

تأییدکننده‌ی بایت‌کد، بارکننده‌ی کلاس، مدیر امنیت، کنترل‌کننده دسترسی، مجوزها، سیاست‌ها، و دامنه‌های محافظت.

تاییدکننده‌ی بایت‌کد، تشخیص می دهد که بایت‌کدها، از کد خارجی برنامه‌ی کاربری جاوا، به‌عنوان مشخصه‌ای از زبان جاوا، بر پلت‌فرم، بارگذاری می‌شوند. سپس، بارکننده‌ی کلاس، مسئول ترجمه‌ی حقیقی بایت‌کدها به ساختار کلاس جاوا است که می تواند توسط جاوا در زمان اجرا، دستکاری شود. بارکنندگان کلاس مختلفی وانند در فرایند بارگذاری کلاس‌ها، به کار گرفته شوند. از این‌رو، سیاست‌های مختلفی هم تعریف می‌شودکه کلاس‌های مطمئن باید در زمان اجرا، بارگذاری شوند.



شکل ۱-۲: معماری امنیتی هسته‌ی جاوا

بسته‌ی *java.security*، شامل کلاس‌ها و واسط‌هایی است که معماری هسته‌ی امنیتی جاوا را تعریف می‌کنند. بسته‌ی *java.security.acl* نیز، شامل کلاس‌های کنترل دسترسی و واسط‌هایی است که در معماری امنیتی هسته‌ی جاوا 1.1 وجود داشتند، اما با ساختارهای جدیدی از کنترل دسترسی در جاوا 2، جایگزین شده‌اند. سرانجام، سایر کلاس‌های مرتبط با امنیت، در مجموعه‌ی کلی بسته‌های پلت‌فرم جاوا قرار دارند. در این بخش و بخش بعدی، مشخص خواهیم نمود که کدام کلاس‌ها، نقشی در معماری امنیتی هسته‌ی جاوا ایفا می‌نمایند.

۱,۳,۱ معماری رمزنگاری جاوا

معماری رمزنگاری جاوا، زیرساختی را برای اجرای عملکرد رمزنگاری بنیادین پلت‌فرم جاوا فراهم می‌سازد. محدوده‌ی عملکرد رمزنگاری، شامل محافظت از داده‌ها در مقابل فساد، ناشی از جامعیت داده‌ها و با استفاده از رمزنگاری اساسی توابع و الگوریتم‌ها است.

۲,۳,۱ گسترش رمزنگاری جاوا

گاهی، اصطلاحات رمزدار کردن[۵] و رمزنگاری[۶] به جای هم استفاده می‌شوند، رمزنگاری، نمایان‌گر یکپارچگی داده‌ها و توابع شناسه‌ی منبع است که توسط JCA پشتیبانی می‌شوند. رمزدار کردن، به این معنی است که دسته‌ای از توابع، تا زمانی که می‌توانند توسط گیرنده‌ی موردنظر رمزگشایی شوند، برای حفظ محرمانگی، روی بلاک‌های داده‌ای، رمزگذاری شوند. توسعه‌ی رمزنگاری جاوا (JCE)[۷]، به‌عنوان یک افزونه‌ی امنیتی جاوا، جهت کمک به این اهداف رمزگذاری، ارائه شده‌است.

۳,۳,۱ واسط‌های مدیر امنیت

تصویر ۱-۳، کلاس ایزوله‌ی *java.lang.SecurityManager* را، که واسطی کلیدی در مدیریت امنیت حال حاضر فرایند ماشین جاوا است، نشان می‌دهد. متد *java.lang.System.getSecurityManager()*، مقداری را که در حال حاضر به‌دست آمده‌است، از شی کلاس *SecurityManager*، باز می‌گرداند. اگر هیچ مدیر امنیتی تعیین نگردد، یک مقدار *null* بازگردانده می‌شود. متد *java.lang.System.getSecurityManager()*، پارامترهای ورودی کلاس *SecurityManager* را فراخوانی کرده و بررسی می‌کند آیا مدیر امنیت، اجازه دارد با کلاس فراخوانی‌شده جایگزین شود یا خیر. اگر هیچ مدیر امنیتی وجود نداشته و یا اگر این کلاس، اجازه‌ی جایگزینی با مدیر امنیت موجود را داشته‌باشد، عملیات پایان می‌یابد و استثنای *SecurityException* ایجاد می‌گردد.

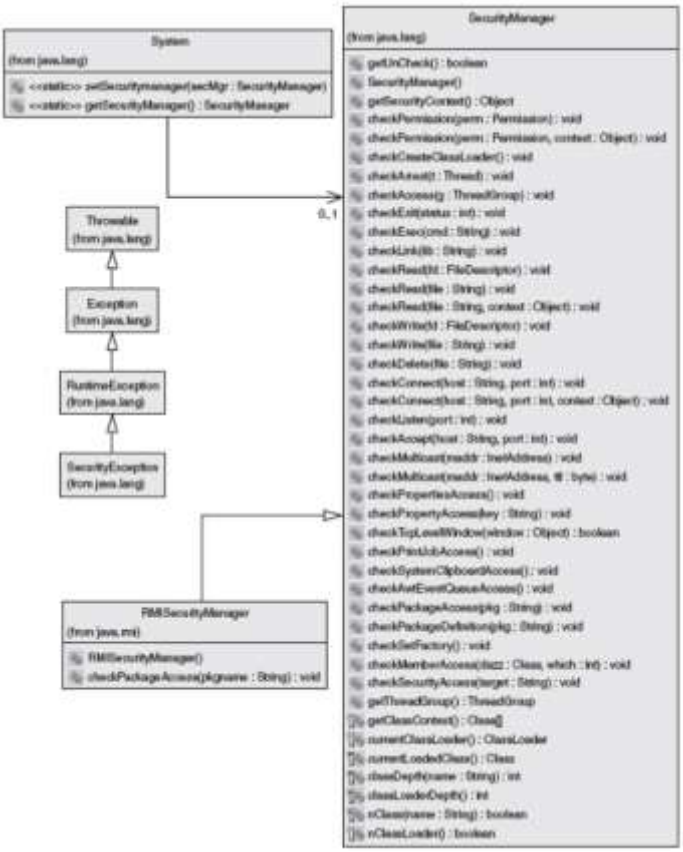
---

[۵] Encryption

[۶] Cryptography

[۷] Java Cryptography Extension (JCE)

کلاس *SecurityManager*، شهرت خود را مدیون متدهای سبک عمومی *chheckXXX()* است. هر متد *chheckXXX()*، امکان دسترسی به مقداری خاص از منابع را بررسی می‌نمایند.



شکل ۱-۳: کلاس‌های مدیر امنیت

۴،۳،۱ مدیران امنیتی سفارشی

جاوا 1.1، به یک کلاس *SecurityManager* گسترش‌یافته نیاز دارد تا برنامه‌ی کابردی خاص شما را با شرایط کنترل دسترسی خاص، ارائه دهد. اگرچه، پیاده‌سازی *SecurityManager* سفارشی، به مدت کوتاهی برای برنامه‌های کاربردی جاوا 1.2 توصیه شد. به‌دلیل آن‌که پلت‌فرم جاوا 2 موجب غیرفعال کردن زیرکلاس‌های *SecurityManager* می‌شود، تمام متدهای محافظت‌شده‌ی کلاس *SecurityManager* – به‌جز *getClassContext()* – در جاوا 2 منسوخ شده‌اند.

کلاس‌هایی مانند *java.rmi.RMISecurityManager*، در نسخه‌ی 1.1 جاوا معرفی شدند. این کلاس‌ها، که در شکل ۱-۳ مشاهده می‌شوند، به‌سادگی، متدهایی را که مرتبط با تغییر رفتار برای بررسی دسترسی

منابع با مقادیر خاص هستند، لغو می‌نمایند. بنابراین، کلاس لغوشده‌ی *SecurityManager* در جاوا 1.1، که چندین عملیات بررسی دسترسی را گسترش می‌دهد، مشابه کد زیر است:

```
public class CustomSecurityManager extends SecurityManager {
	public CustomSecurityManager() {
		super();
	}
	public void checkRead(String fileName) {
		if(fileName != null && fileName.endsWith(“.java”)){
			throw new SecurityException(“ You are not allowed to read “
				+” file names ending with .java”);
		}
		super.checkRead(fileName);
	}
	public void checkWrite(String fileName) {
		if(fileName != null && fileName.endsWith(“.java”)){
			throw new SecurityException(“ You are not allowed to write “
				+” file names ending with .java”);
		}
		super.checkWrite(fileName);
	}
	public void checkDelete(String fileName) {
		if(fileName != null && fileName.endsWith(“.java”)){
			throw new SecurityException(“You are not allowed to delete “
				+” file names ending with .java”);
		}
		super.checkDelete(fileName);
	}
}
```

اگر متد *checkRead()*، کلاس *SecurityException* را فراخوانی کند، متد *myFileAccessMethod()*، پیش از آن‌که باقی متد بتواند ادامه پیدا کند، خاتمه خواهد یافت و با ایجاد *securityException*، بازخواهدگشت، زیرا کلاس *SecurityException* از کلاس *java.lang.RuntimeException* گسترش می‌یابد. در واقع، متد *myFileAccessMethod()* می‌تواند این استثنا را ایجاد کند و نیازی به تعریف صریح امضای متد نیست.

۵,۳,۱ معماری رمزنگاری جاوا

در ابتدا، JCA به‌همراه پلت‌فرم جاوا 1.1 معرفی گردید. JCA، توابع رمزنگاری پایه را، که در ادامه با اهداف زیر مورد استفاده قرار می‌گیرد، فراهم می‌سازد:

- محافظت از یکپارچگی داده‌های ارتباطی یا ذخیره‌شده
- شناسایی اصل مرتبط با داده‌هایی که در حافظه، ذخیره و یا از آن، بازیابی شده‌اند
- ارائه‌ی یک پشتیبانی برای تولید کلیدها و گواهی‌های استفاده‌شده برای شناسایی منابع
- ارائه‌ی یک چارچوب برای پلاگین الگوریتم‌های رمزنگاری، از ارائه‌دهندگان خدمات مختلف

در حقیقت، JCA برای رمزگذاری داده‌های ارتباط داده‌شده و یا ذخیره‌شده مفید نیست. این قابلیت رمزنگاری استفاده‌شده، برای ارائه‌ی محرمانگی توسط JCE، امکان‌پذیر است، زیرا امریکا، صادرات JCE را، که جدا از پلت‌فرم جاوا حمل می‌شود، محدود نموده‌است. این در حالی است که JCA، این محدودیت‌ها را به‌همراه ندارد. علاوه بر این موضوع، SSL (یکی از پروتکل‌های معروف رمزنگاری) در بسته‌ی JCA وجود ندارد، اما واسط SSL، توسط پلت‌فرم جاوا در بسته‌ی JSSE، پشتیبانی می‌شود.

۱,۵,۳,۱ معماری JCA

JCA، ترکیبی از چندین کلاس و واسط است که عملکرد رمزنگاری را پیاده‌سازی می‌کنند. در واقع، بسته‌های پلت‌فرم جاوا 2 زیر، شامل کلاس‌ها و واسط‌های تشکیل‌دهنده‌ی JCA هستند:

- *Java.security*: مجموعه‌ای از کلاس‌های هسته‌ی جاوا و واسط‌های آن، برای چارچوب *plug-and-play* خدمات‌رسان و واسط‌های عملیات رمزنگاری است. توجه داشته‌باشید که این بسته، شامل کلاس‌های هسته‌ی معماری امنیتی جاوا و واسط‌های آن نیز است.
- *Java.security.cert*: مجموعه‌ای از مدیریت گواهی‌ها و واسط‌ها است.
- *Java.security.interfaces*: مجموعه‌ای از واسط‌ها است که به‌منظور ایزوله‌سازی و مدیریت کلیدهای خصوصی و عمومی DSA و RSA استفاده می‌شود.
- *Java.security.spec*: مجموعه‌ای از کلاس‌ها و واسط‌ها است که به‌منظور توصیف کلیدهای خصوصی و عمومی الگوریتم‌ها و پارامترهای خاص استفاده می‌شود.

موتورهای رمزنگاری مختلف (برای مثال *SomeCSPCryptoEngineImpl*) توسط یک CSP، که برخی از استانداردهای رمزنگاری خدمات‌رسان را پیاده‌سازی می کند، پشتیبانی می‌شوند: برای مثال، *ACryptoEngineClassSPI*، توسط JCA فراهم می‌شود. هر واسط خدمات‌رسان، توسط یک موتور

رمزنگاری API، گسترش می‌یابد، که توسط برنامه‌نویس برنامه‌ی کاربردی، مورد استفاده قرار می‌گیرد. هر موتور رمزنگاری API، از یک متد *getInstance(String)* استفاده می‌نماید، که این متد، نام یک الگوریتم خاص مرتبط با کلاس *engine* را دریافت می‌کند. در صورتی‌که موجود باشد، یک نمونه از موتور رمزنگاری درخواستی را نشان می‌دهد. به‌عبارت دیگر، یک *java.security.NoSuchAlgorithmException*، استثنای به‌وقوع پیوسته‌ای است که متد ایستایی را به نام *getInstance(String, String)*، در حافظه‌ی موقت موتور رمزنگاری API، مشخص می کند.

۲,۵,۳,۱ موتورهای رمزنگاری

کلاس انتزاعی *java.security.MessageDigest*، نمونه‌ای از موتور رمزنگاری API است؛ در حالی‌که کلاس انتزاعی *java.security.MessageDigestSpi*، واسط خدمات‌رسانی را که باید پیاده‌سازی شود، نشان می‌دهد. متدهای محافظت‌شده‌ی *MessageDigestSpi*، بصری و مرتبط با CSP پیاده‌سازی‌شده هستند.

۴,۱ تأمین‌کنندگان خدمات رمزنگاری

JCA، که توسط پلتفرم جاوا حمل می‌شود، دارای یک CSP پیش‌فرض است. CSP پیاده‌سازی‌شده‌ی پیش‌فرض، از موتور رمزنگاری و الگوریتم‌های زیر پشتیبانی می‌کند:

- الگوریتم MD5
- الگوریتم SHA-1
- الگوریتم DSA برای امضاها
- الگوریتم DSA برای تولید جفت‌کلیدها
- الگوریتم پارامترهای DSA
- تولیدکننده‌ی کلید DSA
- الگوریتم JKS فروشگاه کلید
- کارخانه‌ی گواهی x.509
- الگوریتم تولیدکننده اعداد تصادفی SHA1PRNG

شما می‌توانید CSP را، به‌سادگی، در زمان اجرای جاوا، به‌عنوان یک فایل JAR یا ZIP، به‌همراه کلاس‌هایی که پیاده‌سازی می‌شوند، نصب کنید. باید فایل *JAVA.SECURITY* را در زیردایرکتوری اصلی، که جاوا در آن نصب شده‌است، در <JavaRootInstall>\lib\security\، پیکربندی نمایید. در این فایل، نام کامل کلاس‌های CSP را، که از کلاس *Provider* گسترش یافته‌اند، وارد کنید. ترتیب اولویت‌ها مشخص می‌کند کدام CSP، در مورد الگوریتمی که جهت استفاده انتخاب شده‌است، به کار گرفته شود. CSPها می‌توانند به‌صورت برنامه‌نویسی‌شده، اضافه یا حذف گردند:

```
// Install SunJCE CSP by first creating a Provider instance
Provider providerSunJce = new com.sun.crypto.provider.SunJCE();
// Then add provider using Security class and obtain preference number
int providerPreferenceJCE = Security.addProvider(providerSunJce);

// Or set the preference number yourself increasing preference numbers
// of any providers already installed at that preference level
Provider providerATI = new com.assuredtech.security.JCAProvider();

int providerPreferenceATI = Security.insertProviderAt(providerATI, 1);
// A provider can be removed dynamically using Security class
// For example, to remove default Sun CSP
Securitv.removeProvider("SUN"):
```

۵,۱ جمع‌بندی

معماری امن جاوا، یک واسط استاندارد را برای توسعه‌دهندگان جاوا فراهم می‌کند تا برنامه‌های کاربردی امن تولید شوند. معماری امنیتی هسته‌ی جاوا، ترکیبی از یک تشخیص‌گر بایت‌کد، یک یا چند بارکننده‌ی کلاس، و مدیریت امن چارچوب کنترل دسترسی است. تشخیص‌گر بایت‌کد از بررسی اهداف مخرب سطح پایین پشتیبانی می‌کند. علاوه بر محافظت در اصطلاحات، صداقت کدهای قابل اعتماد را هم بررسی می‌نماید. البته، مدیر امنیت و کنترل دسترسی، مسئول درستی است.

همچنین، بخشی از پلت‌فرم جاوا 2، JCA خدمات‌رسان *plug-and-play* مبتنی بر محافظت از داده‌ها، جامعیت، و شناسایی را فراهم می‌نماید. خروجی پلت‌فرم جاوا، مانند JCE و JSSE، از محرمانگی محافظت می‌نمایند. این در حالی است که JAAS، میزان صداقت را افزایش می‌دهد. اجزای فعلی، با توجه به معماری امنیتی، از نظر امنیت عمومی، توسط جاوا پشتیبانی نمی‌شوند. هر واسط استاندارد، جهت بررسی و حسابرسی امنیتی، دارای نواقصی است.

فصل دوم: امنیت

زبان برنامه‌نویسی جاوا و سیستم زمان‌اجرا، با هدف تأمین تمهیدات امنیتی، طراحی شدند. به‌عنوان نمونه، دستکاری اشاره‌گر، ضمنی بوده و از برنامه‌نویس مخفی است و هر تلاشی برای ارجاع به یک اشاره‌گر تهی[۸]، منجر به ایجاد یک استثنا[۹] می‌شود. به طور مشابه، یک استثنا ناشی از هرگونه تلاشی برای دستیابی به یک آرایه یا رشته خارج از محدوده‌ی آن است. جاوا زبانی است که وابستگی زیادی به نوع دارد[۱۰]، و تمام تبدیلات نوع، در آن به‌خوبی تعریف شده‌اند و مستقل از پلتفرم هستند. این موضوع در مورد انواع محاسباتی و تبدیلات نیز صدق می‌کند. ماشین مجازی جاوا (JVM)، یک بایت‌کد برای اعتبارسنجی[۱۱] دارد تا اطمینان حاصل نماید بایت‌کدی که در حال اجرا است، با مشخصات زبان جاوا (Java SE 7 Edition (JLS[۱۲])) تطابق دارد، تا از این‌روی، تمام بررسی‌های تعریف‌شده در زبان در جای خود قرار گیرند و نتوان آنها را دور زد.

بارگذار کلاس جاوا[۱۳]، کلاس‌ها را با بار کردن آنها در JVM، شناسایی می‌کند و می‌تواند بین کلاس‌های سیستم مورد اعتماد و سایر کلاس‌هایی، که ممکن است مورد اعتماد نباشند، تمایز قائل شود. می‌توان با امضا کردن دیجیتالی کلاس‌هایی از منابع خارجی، به آنها اولویت و امتیاز[۱۴] اعطا نمود؛ این امتیازهای دیجیتال نیز می‌توانند توسط بارگذار کلاس، مورد بررسی قرار گیرند و به شناسایی کلاس کمک کنند. همچنین، جاوا یک ساز و کار قابل انعطاف دانه‌ریز برای امنیت فراهم می‌کند که برنامه‌نویس را قادر می‌سازد تا دسترسی به منابعی مانند اطلاعات سیستم، فایل‌ها، سوکت‌ها، و هر منبع دیگری را که از لحاظ امنیتی حساس است و برنامه‌نویس تمایل دارد از آن استفاده نماید، کنترل کند. ممکن است این ساز و کار امنیتی نیاز داشته‌باشد که از مدیریت امنیتی در زمان اجرا برخوردار باشد تا سیاست امنیت را اعمال نماید. معمولاً، یک مدیریت امنیت و سیاست امنیت متعلق به آن، توسط آرگومان‌های خط فرمان مشخص

---

[۸] Null
[۹] Exception
[۱۰] Strongly typed
[۱۱] Bytecode verifier
[۱۲] Java Language Specification
[۱۳] Java class loader
[۱۴] Privilege

می‌شوند، اما ممکن است با برنامه‌نویسی نصب شوند (در صورتی‌که چنین عملی توسط یک سیاست امنیتی موجود ممنوع نشده‌باشد). این امکان وجود دارد که امتیازات دسترسی به منابع، با تکیه بر شناسایی مهیاشده توسط ساز و کار بارگذار کلاس، به کلاس‌های جاوای غیرسیستمی، گسترش داده شوند.

برنامه‌های کاربردی سازمانی جاوا[۱۵]، به‌دلیل آن‌که داده‌های غیر قابل اعتماد را می‌پذیرند و با زیرسیستم‌های پیچیده تعامل دارند، مستعد حمله هستند. حملات تزریق (مانند XSS[۱۶]، Xpath و LDPA)، هنگامی‌که مولفه‌هایی که مستعد این حملات هستند در برنامه استفاده شوند، محتمل هستند. یک متد موثر برای کاهش این امر، ساخت لیست‌سفید برای ورودی، و رمزگذاری یا رهایی[۱۷] خروجی، پیش از آن‌که برای تحویل پردازش شود، است.

این فصل، حاوی خط مشی‌هایی در ارتباط با حصول اطمینان از برنامه‌های کاربردی مبتنی بر جاوا بوده که شامل موارد زیر است:

- برخورد با داده‌های حساس
- اجتناب از حملات تزریق رایج
- ویژگی‌های زبان که بتوان از آنها برای به‌خطر انداختن امنیت سوءاستفاده کرد.
- جزئیات ساز و کار امنیت دانه‌ریز جاوا

1.2 محدود کردن طول عمر داده‌های حساس

داده‌های حساس در حافظه نیز می‌توانند نسبت به افشا، آسیب‌پذیر باشند. مهاجمی که می‌تواند کد را به‌عنوان برنامه‌ی کاربردی، روی همان سیستم اجرا کند، در صورتی‌که شرایط مطرح‌شده در ذیل برای برنامه‌ی کاربردی صدق کند، شاید قادر باشد تا به چنین داده‌هایی دست یابد:

---

[۱۵] Enterprise Java applications

[۱۶] Cross-site scripting

[۱۷] Escape

- برای ذخیره‌سازی داده‌های حساس، از اشیایی استفاده می‌کند که محتویات آنها پس از استفاده، پاک یا زباله‌ی جمع‌آوری‌شده[۱۸] نیست.
- دارای صفحاتی از حافظه است که مطابق با نیاز سیستم‌عامل (مثلاً برای انجام وظایف مدیریت حافظه یا پشتیبانی از هایبرنت شدن[۱۹])، می‌توانند از دیسک بیرون آورده شوند.
- داده‌های حساس، مانند BufferedReader را، که کپی‌هایی از داده‌ها را در حافظه‌ی نهان[۲۰] سیستم‌عامل یا در حافظه نگه می‌دارد، در بافر حفظ می‌کند.
- جریان کنترلی خود را بر مبنای انعکاسی می‌گذارد که امکان اقدامات متقابل را برای دور زدن محدودسازی طول عمر متغیرهای حساسف فراهم می‌کند.
- داده‌های حساس را در پیام‌های اشکال‌زدایی[۲۱]، فایل‌های لاگ، متغیرهای محیطی، یا از طریق دامپ‌های هسته و نخ[۲۲]، آشکار می‌سازد.

اگر حافظه حاوی داده‌هایی باشد که پس از استفاده، پاک نشده‌باشند، احتمال نشت داده‌های حساس افزایش می‌یابد. برنامه‌ها باید به‌منظور محدود نمودن خطر افشا، طول عمر داده‌های حساس را محدود کنند.

رفع کامل (یعنی محافظت کامل و بدون خطا از داده‌های حافظه)، نیازمند پشتیبانی از سیستم‌عامل زیربنایی و JVM است. به‌عنوان نمونه، اگر مساله، انتقال داده‌های حساس به بیرون و روی دیسک باشد، به یک سیستم‌عامل امن نیاز است که خروج و هایبرنت شدن را غیرفعال کند.

---

[۱۸] Garbage-collected
[۱۹] Hibernation
[۲۰] Cache
[۲۱] Debugging
[۲۲] Thread and core dumps

۱,۱,۲ نمونه‌کد ناسازگار

این نمونه کد ناسازگار، اطلاعات نام کاربری و رمز عبور را از کنسول می‌خواند و رمز عبور را به‌صورت یک شی *String*، ذخیره می‌کند. مدارک، تا زمانی‌که جمع‌کننده‌ی زباله[۲۳]، حافظه‌ی مربوط به رشته را پس بگیرد، در معرض افشا باقی می‌مانند.

```
class Password {
	public static void main (String args[]) throws IOException {
		Console c = System.console();
		if (c == null) {
			System.err.println("No console.");
			System.exit(1);
		}
		String username = c.readLine("Enter your user name: ");
		String password = c.readLine("Enter your password: ");
		if (!verify(username, password)) {
			throw new SecurityException("Invalid Credentials");
		}
		// ...
	}
	// Dummy verify method, always returns true
	private static final boolean verify(String username,String password) {
		return true;
	}
}
```

۲,۱,۲ راه‌حل سازگار

این راه‌حل، از متد *Console.readPassword()* برای گرفتن رمز عبور از کنسول استفاده می‌کند.

```
class Password {
	public static void main (String args[]) throws IOException {
		Console c = System.console();
		if (c == null) {
			System.err.println("No console.");
			System.exit(1);
		}
		String username = c.readLine("Enter your user name: ");
		char[] password = c.readPassword("Enter your password: ");
		if (!verify(username, password)) {
			throw new SecurityException("Invalid Credentials");
		}
		// Clear the password
		Arrays.fill(password, ' ');
	}
	// Dummy verify method, always returns true
	private static final boolean verify(String username,char[] password) {
		return true;
	}
}
```

متد *Console.readPassword()*، منجر به برگرداندن رمز عبور به‌صورت دنباله‌ای از کاراکترها (به جای *String*) می‌شود. در نتیجه، برنامه‌نویس می‌تواند رمز عبور را بلافاصله پس از استفاده، از آرایه پاک کند. این متد، انعکاس[۲۴] رمز عبور به کنسول را غیرفعال می‌کند.

۳.۱.۲ نمونه‌کد ناسازگار

این نمونه‌کد، از *BufferedReader* برای در لفافه قرار دادن یک شیِ *InputStream-Reader* استفاده می‌کند تا بتوان داده‌های حساس را از یک فایل خواند:

```
void readData() throws IOException {
	BufferedReader br = new BufferedReader(new InputStreamReader(
		new FileInputStream("file")));
	// Read from the file
	String data = br.readLine();
}
```

می‌تواند پس از آن‌که دیگر به داده نیازی نیست، باقی بماند. متد *BufferedReader.read(char[], int, int)* می‌تواند یک آرایه‌ی کاراکتری را بخواند و اشغال کند[۲۵]. با این‌حال، نیاز دارد که برنامه‌نویس به‌صورت

[۲۴] Echoing

[۲۵] Populate

دستی، داده‌های حساس را پس از استفاده، از آرایه پاک کند. از سوی دیگر، حتی اگر *BufferedReader* بخواهد یک شی *FileReader* را پنهان سازد، از همان نقاط ضعف رنج خواهد برد.

۴,۱,۲ راه‌حل سازگار

این راه‌حل، از یک بافر NIO[۲۶]، که مستقیما تخصیص داده شده‌است، برای خواندن داده‌های حساس از یک فایل استفاده می‌کند. داده می‌تواند بلافاصله پس از استفاده پاک شود و در چندین مکان، در حافظه‌ی نهان قرار نگیرد و بافر نشود، و تنها در حافظه‌ی سیستم وجود داشته‌باشد.

```
void readData()
{
	ByteBuffer buffer = ByteBuffer.allocateDirect(16 * 1024);
	try (FileChannel rdr = (new FileInputStream("file")).getChannel())
	{
		while (rdr.read(buffer) > 0)
		{
			// Do something with the buffer
			buffer.clear();
		}
	}
	catch (Throwable e)
	{
		// Handle error
	}
}
```

توجه داشته‌باشید که پاک کردن داده‌های بافر، الزامی است، زیرا بافرها مستقیما زباله‌روبی نمی‌شوند.

۵,۱,۲ کاربرد

شکست در محدود کردن طول عمر داده‌های حساس می‌تواند منجر به نشت اطلاعات شود.

---

[۲۶] New I/O

۲,۲ داده‌های حساس رمزنگاری‌نشده را در سمت مشتری ذخیره نکنید

هنگام ساخت یک برنامه‌ی کاربردی که از مدل مشتری-خدمت‌گزار استفاده می‌کند، در صورتی‌که مشتری مستعد حمله باشد، ذخیره‌سازی اطلاعات حساس (مانند گواهی‌های کاربر) در سمت مشتری می‌تواند منجر به افشای غیر مجاز شود. رایج‌ترین متد کاهش این مساله در برنامه‌های کاربردی وب، فراهم نمودن یک کوکی[۲۷] برای مشتری و ذخیره‌سازی اطلاعات حساس روی سرور است. کوکی‌ها توسط یک وب‌سرور ساخته می‌شوند و برای یک دوره‌ی زمانی، روی مشتری ذخیره می‌گردند. هنگامی‌که مشتری مجدداً به سرور متصل می‌شود، کوکی برای آن مهیا می‌شود، که مشتری را به سرور می‌شناساند. پس از آن، سرور، اطلاعات حساس را ارائه می‌کند.

کوکی‌ها از اطلاعات حساس در برابر حملات XSS[۲۸] محافظت نمی‌کنند. یک مهاجم که قادر به کسب یک کوکی یا از طریق حمله‌ی XSS، و یا با حمله‌ی مستقیم به مشتری است، می‌تواند اطلاعات حساس را با استفاده از کوکی، از سرور بگیرد. اگر سرور پس از سپری شدن یک مدت محدود، مانند ۱۵ دقیقه،

---

[۲۷] Cookie

[۲۸] Cross-site scripting attacks

نشست[۲۹] را نامعتبر و باطل کند، این خطر دارای مهلت زمانی[۳۰] است. معمولاً یک کوکی، یک رشته‌ی کوتاه است. اگر حاوی اطلاعات حساس باشد، باید آن اطلاعات رمزنگاری شوند. اطلاعات حساس، شامل نام‌های کاربری، رمزهای عبور، شماره‌های کارت‌های اعتباری، شماره‌های امنیت اجتماعی[۳۱]، و هر اطلاعات شناسایی شخصی دیگری در مورد کاربر است.

۱.۲.۲ نمونه‌کد ناسازگار

در این نمونه‌کد، ورود به سرولت[۳۲]، نام کاربری و رمز عبور را در کوکی ذخیره می‌کند تا کاربر را برای درخواست‌های متعاقب، شناسایی نماید:

```
protected void doPost(HttpServletRequest request,HttpServletResponse response)
{
	// Validate input (omitted)
	String username = request.getParameter("username");
	char[] password = request.getParameter("password").toCharArray();
	boolean rememberMe =Boolean.valueOf(request.getParameter("rememberme"));
	LoginService loginService = new LoginServiceImpl();
	if (rememberMe) {
		if (request.getCookies()[0] != null &&
			request.getCookies()[0].getValue() != null) {
			String[] value =
				request.getCookies()[0].getValue().split(";");
			if (!loginService.isUserValid(value[0],
				value[1].toCharArray())) {
				// Set error and return
			}
			else {
				// Forward to welcome page
			}
		}
		else {
		  boolean validated=loginService.isUserValid(username, password);
		  if (validated)   {
			Cookie loginCookie = new Cookie("rememberme", username +
				";" + new String(password));
			response.addCookie(loginCookie);
			// ... forward to welcome page
		  }
		  else   {
			// Set error and return
		  }
		}
	}
	else {
		// No remember-me functionality selected Proceed with regular
		authentication;
		// if it fails set error and return
	}
	Arrays.fill(password, ' ');
}
```

با این‌حال، تلاش برای پیاده‌سازی عملکرد "به خاطر سپاری[۳۳]"، ناامن است، زیرا یک مهاجم، با دسترسی به ماشین مشتری می‌تواند این اطلاعات را مستقیما از مشتری به‌دست آورد. این کد، "ذخیره‌سازی رمزهای عبور با استفاده از تابع درهم‌سازی" را نیز نقض می‌کند.

2.2.2 راه‌حل سازگار (نشست)

این راه‌حل، عملکرد به خاطر سپاری را با ذخیره‌سازی نام کاربری و یک رشته‌ی تصادفی امن در کوکی، پیاده‌سازی می‌کند. همچنین، با استفاده از *HttpSession*، وضعیت نشست را نگه می‌دارد:

سرور، نگاشتی میان نام‌های کاربری و رشته‌های تصادفی امن نگه می‌دارد. هنگامی‌که یک کاربر "مرا به خاطر بسپار" را انتخاب می‌کند، متد *doPost()* بررسی می‌نماید که آیا کوکی‌های فراهم‌شده، حاوی جفت نام کاربری مجاز و رشته‌ی تصادفی هستند یا خیر. اگر نگاشت، حاوی یک جفت منطبق باشد، سرور،

[۳۳] Remember-me

کاربر را احراز اصالت کرده و او را به صفحه‌ی خوش‌آمدگویی می‌فرستد. در غیر این‌صورت، سرور، یک خطا به مشتری بر می‌گرداند. اگر کاربر "مرا به خاطر بسپار" را انتخاب کند، اما مشتری نتواند یک کوکی مجاز فراهم نماید، سرور از کاربر می‌خواهد تا با استفاده از گواهی‌های خود، احراز اصالت کند. اگر احراز اصالت موفقیت‌آمیز بود، سرور یک کوکی جدید با ویژگی‌های به خاطر سپاری صادر می‌نماید.

این راه‌حل، از طریق غیرمعتبر کردن نشست فعلی و ساخت یک نشست جدید، از حملات تثبیت جلسه[۳۴] جلوگیری به‌عمل می‌آورد. همچنین، پنجره‌ای را، که طی آن یک مهاجم می‌تواند یک حمله‌ی نشست‌ربایی[۳۵] انجام دهد، با تنظیم مهلت نشست به ۱۵ دقیقه بین دسترسی‌های مشتری، کاهش می‌دهد.

```
protected void doPost(HttpServletRequest request,HttpServletResponse response)
{
	// Validate input (omitted)
	String username = request.getParameter("username");
	char[] password = request.getParameter("password").toCharArray();
	boolean rememberMe =Boolean.valueOf(request.getParameter("rememberme"));
	LoginService loginService = new LoginServiceImpl();
	boolean validated = false;
	if (rememberMe) {
		if (request.getCookies()[0] != null &&
			request.getCookies()[0].getValue() != null) {
		   String[] value = request.getCookies()[0].getValue().split(";");
		   if (value.length != 2) {
			// Set error and return
		   }
		   if (!loginService.mappingExists(value[0], value[1])) {
			// (username, random) pair is checked. Set error and return
		   }
		}
		else {
			validated = loginService.isUserValid(username, password);
			if (!validated) {
				// Set error and return
			}
		}
		String newRandom = loginService.getRandomString();
		// Reset the random every time
		loginService.mapUserForRememberMe(username, newRandom);
		HttpSession session = request.getSession();
		session.invalidate();
		session = request.getSession(true);
		// Set session timeout to 15 minutes
		session.setMaxInactiveInterval(60 * 15);
		// Store user attribute and a random attribute in session scope
		session.setAttribute("user", loginService.getUsername());
		Cookie loginCookie =new Cookie("rememberme", username + ";" +
			newRandom);
		response.addCookie(loginCookie);
```

۳.۲.۲ کاربرد

ذخیره‌سازی اطلاعات حساس در سمت مشتری، منجر به در دسترس بودن این اطلاعات برای هر کسی که بتواند به مشتری حمله کند، می‌شود.

۳,۲ کلاس‌های حساس قابل تغییر را توسط پنهان‌سازی‌های غیر قابل اصلاح ارائه کنید

تغییرناپذیری فیلدها مانع تغییرات ناخواسته و دستکاری مخربانه[۳۶] می‌شود، به‌طوری‌که در حین پذیرفتن ورودی یا برگرداندن مقادیر، کپی کردن تدافعی[۳۷]، غیر ضروری می‌شود. با این‌وجود، برخی از کلاس‌های حساس را نمی‌توان تغییرناپذیر نمود. خوشبختانه، دسترسی فقط خواندنی به کلاس‌های تغییرپذیر می‌تواند توسط پنهان‌سازی غیرقابل اصلاح، به کد غیرقابل اعتماد اعطا شود. به‌عنوان نمونه، کلاس‌های *Collection*، شامل مجموعه‌ای از پنهان‌سازی‌ها هستند که که به مشتری‌ها اجازه می‌دهند یک دید غیرقابل اصلاح از یک شی *Collection* را مشاهده نمایند.

۱,۳,۲ نمونه‌کد ناسازگار

این مثال، حاوی کلاس *Mutable* است و اجازه می‌دهد شی آرایه‌ی داخلی، اصلاح شود:

```
class Mutable
{
	private int[] array = new int[10];
	public int[] getArray()
	{
		return array;
	}
	public void setArray(int[] i)
	{
		array = i;
	}
}
//...
private Mutable mutable = new Mutable();
public Mutable getMutable() {return mutable;}
```

یک فراخوان[۳۸] غیرقابل اعتماد، متد متقاطع[۳۹] *setArray()* را فراخوانی می‌کند، و ویژگی تغییرناپذیری شی را نقض می‌نماید. همچنین، با فراخوانی متد گیرنده‌ی *getArray()*، اجازه‌ی اصلاح وضعیت خصوصی داخلی کلاس را صادر می‌کند. این کلاس، "OBJ05-J. اعضای کلاس قابل تغییر خصوصی را پیش از بازگرداندن ارجاعات آنها، به‌صورت تدافعی کپی کنید"، را نیز نقض می‌کند.

۲.۳.۲ نمونه‌کد ناسازگار

این مثال، کلاس *Mutable* را از طریق زیرکلاس *MutableProtector* گسترش می‌دهد:

```
class MutableProtector extends Mutable
{
	@Override
	public int[] getArray()
	{
		return super.getArray().clone();
	}
}
// ...
private Mutable mutable = new MutableProtector();
// May be safely invoked by untrusted caller having read ability
public Mutable getMutable() {return mutable;}
```

[۳۸] Invoker

[۳۹] Mutator

در این کلاس، فراخوانی *getArray()*، طبق "OBJ05-J، اجازه‌ی اصلاح وضعیت داخلی خصوصی کلاس را نمی‌دهد. اگرچه، ممکن است یک فراخواننده‌ی غیرقابل اعتماد، متد *setArray()* را فراخوانی کند و شی *Mutable* را تغییر دهد.

۳.۳.۲ راه‌حل سازگار

به‌طور کلی، کلاس‌های حساس می‌توانند با فراهم کردن پنهان‌سازهای مناسب برای تمام متدهای تعریف‌شده توسط واسط هسته، شامل متدهای قابل تغییر، به اشیا نمایش ایمن[۴۰] تبدیل شوند. پنهان‌سازها باید برای تمام متدهای قابل تغییر، یک استثنای *UnsupportedOperationException* ایجاد کنند تا مشتریان نتوانند عملیاتی انجام دهند که بر ویژگی تغییرناپذیری شی اثر بگذارد.

این راه‌حل سازگار، متد *setArray()* را اضافه می‌کند، که متد *Mutable.setArray()* را لغو می‌کند و جلوی تغییر شی *Mutable* را می‌گیرد:

```
class MutableProtector extends Mutable
{
	@Override
	public int[] getArray()
	{
		return super.getArray().clone();
	}
	@Override
	public void setArray(int[] i)
	{
		throw new UnsupportedOperationException();
	}
}
// ...
private Mutable mutable = new MutableProtector();
// May be safely invoked by untrusted caller having read ability
public Mutable getMutable() {return mutable; }
```

[۴۰] Safe-view

کلاس پنهان‌ساز *MutableProtector*، متد *getArray()* را لغو می‌کند و آرایه را کلون[۴۱] می‌نماید. با وجود این‌که، فراخوانی کد، یک کپی از آرایه‌ی شی قابل تغییر را دریافت می‌کند، اما آرایه‌ی اصلی، بدون تغییر و غیرقابل دسترسی باقی می‌ماند. متد *setArray()* که لغو را انجام می‌دهد، در صورتی‌که فراخوانی‌کننده تلاش کند از این متد روی شی بازگشتی استفاده کند، یک استثنا ایجاد می‌نماید. وقتی دسترسی خواندنی داده مجاز است، این شی می‌تواند به کد غیر قابل اعتماد داده شود.

۴،۳،۲ کاربرد

شکست در فراهم‌سازی یک دید غیرقابل اصلاح و ایمن از یک شی قابل تغییر حساس برای یک غیر قابل اعتماد، می‌تواند منجر به تغییر مخربانه و خرابی شی شود.

۴،۲ مطمئن شوید متدهای حساس به امنیت، توسط آرگومان‌های اعتبارسنجی‌شده فراخوانی می‌شوند

کد برنامه‌ی کاربردی که متدهای حساس به امنیت را فراخوانی می‌کند، باید آرگومان‌هایی را که به این متدها داده می‌شوند، اعتبارسنجی نماید. به‌طور مشخص، این امکان وجود دارد مقادیر تهی، توسط متدهای حساس به امنیت خاصی، بی‌خطر درنظر گرفته شوند، اما ممکن است تنظیمات پیش‌فرض را لغو کنند. اگرچه، کد متدهای حساس به امنیت باید با حالت تدافعی نوشته شود، لازم است کد مشتری، آرگومان‌هایی را که احتمال دارد در شرایط غیر، به‌عنوان معتبر پذیرفته شوند، اعتبارسنجی نماید. شکست در انجام این کار می‌تواند منجر به افزایش امتیاز دسترسی و اجرای کد دلخواه شود.

۱،۴،۲ نمونه‌کد ناسازگار

این مثال، متد *doPrivileged()* حاوی دو آرگومان را نشان می‌دهد که یک مفهوم کنترل دسترسی را به‌عنوان آرگومان دوم می‌گیرد. این کد، امتیازات را از مفهومی که قبلاً ذخیره شده‌بود، بازیابی می‌کند.

```
AccessController.doPrivileged(new PrivilegedAction<Void>()
                                {
                                      public Void run()
                                      {
                                            // ...
                                      }
                                }, accessControlContext);
```

هنگامی که یک مفهوم کنترل دسترسی تهی فرستاده می‌شود، متد دو آرگومانی *doPrivileged()*، در راستای کاهش امتیازات فعلی به مفهوم ذخیره‌شده‌ی قبلی، شکست می‌خورد. در نتیجه، این کد می‌تواند هنگامی‌که آرگومان *accessControlContext* تهی است، امتیازات اضافی اعطا کند. برنامه‌نویسانی که قصد فراخوانی *AccessController.doPrivileged()* را با یک مفهوم کنترل دسترسی تهی دارند، باید به‌وضوح، ثابت تهی را انتقال دهند یا از نسخه‌ی تک‌آرگومانی *AccessController.doPrivileged()* استفاده نمایند.

۲.۴.۲ راه‌حل سازگار

این راه‌حل، با حصول اطمینان از تهی نبودن *accessControlContext*، از اعطای امتیازات اضافی جلوگیری به‌عمل می‌آورد.

```
if (accessControlContext == null)
{
	throw new SecurityException("Missing AccessControlContext");
}
AccessController.doPrivileged(new PrivilegedAction<Void>()
                                {
                                        public Void run()
                                        {
                                                // ...
                                        }
                                }, accessControlContext);
```

۳.۴.۲ کاربرد

متدهای حساس به امنیت باید به‌طور کامل فهمیده شوند و پارامترهای آنها، اعتبارسنجی گردد تا جلوی مواردی با مقادیر آرگومان غیرمنتظره (مانند آرگومان تهی) گرفته شود. اگر مقادیر آرگومان غیرمنتظره به متدهای حساس به امنیت فرستاده شوند، اجرای کد دلخواه و افزایش امتیازهای دسترسی امکان‌پذیر خواهد بود.

۵.۲ از آپلود فایل دلخواه، جلوگیری کنید

برنامه‌های جاوا، از جمله برنامه‌های کاربردی وب، که آپلود فایل را می‌پذیرند، باید اطمینان یابند که یک مهاجم نتواند فایل‌های مخرب را آپلود کند یا انتقال دهد. اگر یک فایل محدودشده، که حاوی کد است، توسط سیستم هدف اجرا شود، می‌تواند پدافندهای لایه‌ی کاربرد را به خطر اندازد. به‌عنوان نمونه، ممکن است برنامه‌ای که آپلود فایل‌های HTML را مجاز می‌داند، به یک کد مخرب اجازه‌ی اجرا دهد؛ مهاجم می‌تواند یک فایل HTML مجاز را، با واحد عملیات[۴۲] XSS، که در غیاب یک روتین رهایی خروجی اجرا خواهد شد، ارسال کند. از این‌رو، بسیاری از برنامه‌ها، نوع فایلی را که می‌توان آپلود کرد، محدود می‌سازند.

همچنین، ممکن است آپلود فایل‌هایی با پسوندهای خطرناکی مانند *exe.* و *sh.*، که می‌توانند منجر به اجرای کد دلخواه روی برنامه‌های سمت سرور شوند، امکان‌پذیر باشد. برنامه‌ای که تنها، فیلد *Content-*

[۴۲] Payload

Type را در سرآیند HTTP محدود می‌کند، مستعد چنین حملاتی است. به‌منظور پشتیبانی از آپلود فایل، یک صفحه‌ی JSP[۴۳] معمولی از کدی مانند مثال زیر تشکیل می‌شود:

```
<s:form action="doUpload" method="POST" enctype="multipart/form-data">
	<s:file name="uploadFile" label="Choose File" size="40" />
	<s:submit value="Upload" name="submit" />
</s:form>
```

بسیاری از چارچوب‌های سازمانی جاوا، تنظیمات پیکربندی را به‌منظور دفاع در برابر آپلود فایل دلخواه، فراهم می‌کنند. متاسفانه، بیشتر آنها در فراهم کردن محافظت کافی، شکست می‌خورند. کاهش این آسیب‌پذیری، شامل بررسی سایز فایل، نوع محتوا، و محتویات فایل، از بین دیگر ویژگی‌های فراداده‌ها[۴۴] است.

۱.۵.۲ نمونه‌کد ناسازگار

این مثال، یک کد XML ناشی از آپلود برنامه‌ی Struts 2 را نشان می‌دهد. کد رهگیر[۴۵]، مسئول آپلودهای فایل زیر است.

```
<action name="doUpload" class="com.example.UploadAction">
	<interceptor-ref name="fileUpload">
		<param name="maximumSize"> 10240 </param>
		<param name="allowedTypes">
			text/plain,image/JPEG,text/html
		</param>
	</interceptor-ref>
</action>
```

کد مربوط به آپلود فایل، در کلاس *UploadAction* است:

```
public class UploadAction extends ActionSupport
{
	private File uploadedFile;
	// setter and getter for uploadedFile
	public String execute()
	{
		try
		{
			// File path and file name are hardcoded for illustration
			File fileToCreate = new File("filepath", "filename");
			// Copy temporary file content to this file
			FileUtils.copyFile(uploadedFile, fileToCreate);
			return "SUCCESS";
		}
		catch (Throwable e)
		{
			addActionError(e.getMessage());
			return "ERROR";
		}
	}
}
```

مقدار نوع پارامتر *maximumSize*، از این‌که *Action* نتواند یک فایل بسیار بزرگ را دریافت نماید، اطمینان حاصل می‌کند. پارامتر *allowedTypes*، نوع فایل‌هایی را که پذیرفته می‌شوند، تعریف می‌کند. با این‌وجود، این متد، در حصول اطمینان از این‌که فایل‌های آپلودشده مطابق با نیازمندی‌های امنیتی باشند، شکست می‌خورد، زیرا بررسی‌های رهگیر می‌توانند به راحتی دور زده شوند. اگر یک مهاجم از یک ابزار پراکسی[۴۶] برای تغییر نوع محتوا در درخواست خام در حال انتقال HTTP استفاده کند، چارچوب، در جلوگیری از آپلود فایل، با شکست مواجه می‌شود. در نتیجه، مهاجم می‌تواند یک فایل مخرب (مثلاً با پسوند exe.) را آپلود نماید.

۲.۵.۲ راه‌حل سازگار

آپلود فایل، تنها باید زمانی پیش برود که نوع محتوا با محتوای حقیقی فایل همخوانی داشته‌باشد. به‌عنوان نمونه، یک فایل با یک سرآیند تصویر، تنها باید حاوی یک تصویر بوده و نباید حاوی کد اجرایی باشد. این راه‌حل سازگار، از کتابخانه‌ی Apache Tika برای کشف و استخراج محتوای فراداده‌ها و متن ساخت‌یافته

[۴۶] Proxy

از مستندات، با استفاده از کتابخانه‌های مفسر[۴۷]، استفاده می‌کند. متد *checkMetaData()* باید پیش از فراخوانی کد در *execute()*، که مسئول آپلود فایل است، فراخوانی شود. *AutoDetectParser*، بهترین مفسر موجود را برمبنای نوع محتوای فایلی که قرار است تفسیر شود، انتخاب می‌کند.

۳.۵.۲ کاربرد

آسیب‌پذیری یک آپلود فایل دلخواه می‌تواند منجر به افزایش امتیاز دسترسی و اجرای کد دلخواه شود.


```
public class UploadAction extends ActionSupport {
	private File uploadedFile;
	// setter and getter for uploadedFile
	public String execute() {
		try {
			// File path and file name are hardcoded for illustration
			File fileToCreate = new File("filepath", "filename");
			boolean textPlain=checkMetaData(uploadedFile,"text/plain");
			boolean img = checkMetaData(uploadedFile, "image/JPEG");
			boolean textHtml = checkMetaData(uploadedFile,"text/html");
			if (!textPlain || !img || !textHtml) {
				return "ERROR";
			}
			// Copy temporary file content to this file
			FileUtils.copyFile(uploadedFile, fileToCreate);
			return "SUCCESS";
		} catch (Throwable e) {
			addActionError(e.getMessage());
			return "ERROR";
		}
	}
	public static boolean checkMetaData(File f, String getContentType) {
		try (InputStream is = new FileInputStream(f)) {
			ContentHandler contenthandler = new BodyContentHandler();
			Metadata metadata = new Metadata();
			metadata.set(Metadata.RESOURCE_NAME_KEY, f.getName());
			Parser parser = new AutoDetectParser();
			try {
				parser.parse(is,contenthandler,metadata,new
				ParseContext());
			}
			catch (SAXException | TikaException e) {
				// Handle error
				return false;
			}
			if (metadata.get(Metadata.CONTENT_TYPE).equalsIgnoreCase(
			getContentType)) {
				return true;
			}
```

۶,۲ خروجی را به‌طور مناسب، رمزنگاری یا رها کنید

پاک‌سازی مناسب ورودی می‌تواند جلوی اضافه کردن داده‌های مخرب به یک زیرسیستم، مانند پایگاه داده، را بگیرد. با این‌حال، زیرسیستم‌های متفاوت، به انواع پاک‌سازی متفاوتی نیاز دارند. خوشبختانه، معمولاً واضح است که کدام زیرسیستم در نهایت، ورودی‌ها را دریافت خواهد کرد و متعاقباً چه نوعی از پاک‌سازی موررد نیاز است.

چندین زیرسیستم برای هدف دادن داده‌ها به خروجی[۴۸] وجود دارند. ارائه‌دهنده‌ی[۴۹] HTML، یکی از زیرسیستم‌های رایج برای نمایش خروجی است. ممکن است به نظر برسد که داده‌های فرستاده‌شده به یک زیرسیستم خروجی، از یک منبع قابل اعتماد نشأت گرفته‌اند. با این‌وجود، فرض غیرضروری بودن پاک‌سازی خروجی، خطرناک است، زیرا ممکن است چنین داده‌هایی به‌صورت غیرمستقیم از یک منبع

[۴۸] Outputting

[۴۹] Renderer

غیرقابل اعتماد نشأت گرفته و حاوی محتویات مخربانه باشند. شکست در پاک‌سازی مناسب داده‌های ارسال‌شده به یک زیرسیستم خروجی، می‌تواند منجر به چندین نوع حمله شود. به‌عنوان نمونه، ارائه‌دهنده‌ی HTML، مستعد حملات تزریق HTML و XSS است. پاک‌سازی خروجی به‌منظور جلوگیری از چنین حملاتی، به اندازه‌ی پاک‌سازی ورودی، حائز اهمیت است.

با اعتبارسنجی ورودی، داده‌ها باید پیش از پاک‌سازی ویژگی‌های مخرب، نرمال شوند. تمام ویژگی‌های خروجی را، به جز آنهایی که ایمن هستند، رمزنگاری کنید تا جلوی آسیب‌پذیری‌های ناشی از داده‌هایی که اعتبارسنجی را دور می‌زنند، گرفته شود (IDS01-J. رشته‌ها را پیش از اعتبارسنجی آنها، نرمال‌سازی کنید).

۱.۶.۲ نمونه‌کد ناسازگار

این مثال، از مفهوم MVC [۵۰] چارچوب Spring EE-based جاوا، برای نمایش داده‌ها به کاربر، بدون رمزنگاری یا رهایی آن، استفاده می‌کند. به‌دلیل آن‌که داده‌ها به یک مرورگر وب فرستاده می‌شوند، کد، هدف هر دو نوع حمله‌ی تزریق HTML و XSS قرار می‌گیرد.

```
@RequestMapping("/getnotifications.htm")
public ModelAndView getNotifications(
        HttpServletRequest request, HttpServletResponse response)
{
        ModelAndView mv = new ModelAndView();
        try
        {
                UserInfo userDetails = getUserInfo();
                List<Map<String,Object>> list =
                        new ArrayList<Map<String, Object>>();
                List<Notification> notificationList =
                        NotificationService.getNotificationsForUserId(
                                userDetails.getPersonId());
                for (Notification notification: notificationList)
                {
                        Map<String,Object> map = new HashMap<String, Object>();
                        map.put("id", notification.getId());
                        map.put("message", notification.getMessage());
                        list.add(map);
                }
                mv.addObject("Notifications", list);
        }
        catch (Throwable t)
        {
                // Log to file and handle
        }
        return mv;
}
```

۲.۶.۲ راه‌حل سازگار

این راه‌حل، یک کلاس *ValidateOutput* را تعریف می‌کند، که خروجی را به یک مجموعه کاراکتر شناخته‌شده، نرمال‌سازی می‌نماید. این کلاس، با استفاده از یک لیست‌سفید، پاک‌سازی خروجی را انجام می‌دهد و هر مقادیر داده‌ای نامشخصی را رمزنگاری می‌نماید تا مکانیزم دوبار بررسی را اعمال نماید. توجه داشته باشید که الگوهای قرار دادن در لیست‌سفید، طبق نیازهای مشخص فیلدهای متفاوت، تغییر خواهند کرد.

```
public class ValidateOutput
{
	// Allows only alphanumeric characters and spaces
	private static final Pattern pattern =
		Pattern.compile("^[a-zA-Z0-9\\s]{0,20}$");
	// Validates and encodes the input field based on a whitelist
	public String validate(String name, String input)
	throws ValidationException
	{
		String canonical = normalize(input);
		if (!pattern.matcher(canonical).matches())
		{
			throw new ValidationException("Improper format in " +
				name + " field");
		}
		// Performs output encoding for nonvalid characters
		canonical = HTMLEntityEncode(canonical);
		return canonical;
	}
	// Normalizes to known instances
	private String normalize(String input)
	{
		String canonical = java.text.Normalizer.normalize(input,
			Normalizer.Form.NFKC);
		return canonical;
	}
	// Encodes nonvalid data
	private static String HTMLEntityEncode(String input)
	{
		StringBuffer sb = new StringBuffer();
		for (int i = 0; i < input.length(); i++)
		{
			char ch = input.charAt(i);
```

```
// ...
@RequestMapping("/getnotifications.htm")
public ModelAndView getNotifications(HttpServletRequest request,
HttpServletResponse response)
{
	ValidateOutput vo = new ValidateOutput();
	ModelAndView mv = new ModelAndView();
	try
	{
		UserInfo userDetails = getUserInfo();
		List<Map<String,Object>> list =
		new ArrayList<Map<String,Object>>();
		List<Notification> notificationList =
			NotificationService.getNotificationsForUserId(
				serDetails.getPersonId());
			for (Notification notification: notificationList)
			{
				Map<String,Object> map = new
				HashMap<String,Object>();
				map.put("id", vo.validate("id"
					,notification.getId()));
				map.put("message",vo.validate("message",
					notification.getMessage()));
				list.add(map);
			}
			mv.addObject("Notifications", list);
	}
	catch (Throwable t)
	{
		// Log to file and handle
	}
	return mv;
```

رمزنگاری و رهانمودن خروجی، هنگام پذیرش کاراکترهای خطرناکی مانند علامت های (") و (<، >)، امری الزامی است. حتی هنگامی‌که ورودی در لیست‌سفید قرار می‌گیرد تا چنین کاراکترهایی را نپذیرد، رهاندن خروجی پیشنهاد می‌شود، زیرا سطح دومی از دفاع را فراهم می‌کند. توجه داشته‌باشید که دنباله‌ی رهایی دقیق می‌تواند بسته به این‌که خروجی کجا تعبیه شده‌است، تغییر یابد. به‌عنوان نمونه، ممکن است خروجی غیرقابل اعتماد، در ویژگی مقدار HTML، CSS، URL، یا اسکریپت رخ دهد؛ در این‌صورت، روتین رمزنگاری خروجی در هر مورد متفاوت خواهد بود. همچنین، استفاده‌ی امن از داده‌های غیرقابل اعتماد در برخی مفاهیم، غیرممکن است.

۳,۶,۲ کاربرد

شکست در رمزنگاری با رهاندن خروجی، پیش از این‌که نمایش داده شود یا از یک مرز اعتماد بگذرد، می‌تواند منجر به اجرای کد دلخواه شود.

۴,۶,۲ آسیب‌پذیری‌های مرتبط

آسیب‌پذیری Apache GERONIMO-1474، که در ماه ژوئن سال ۲۰۰۶ گزارش شد، به مهاجمان اجازه می‌داد URLهایی را که حاوی *JavaScript* بودند، ارسال نمایند. *Web Access Log Viewer*، در پاک‌سازی داده‌هایی که به کنسول مدیر ارسال می‌نمود، شکست خورد. در نتیجه، منجر به یک حمله‌ی کلاسیک XSS گردید.

۷,۲ از تزریق کد جلوگیری کنید

تزریق کد می‌تواند هنگامی رخ دهد که ورودی غیرقابل اعتماد، به کدی که به‌صورت پویا ساخته شده‌است، تزریق شود. یک منبع آشکار از آسیب‌پذیری‌های محتمل، استفاده از JavaScript متعلق به کد جاوا است. پکیج *javax.script*، از واسط‌ها و کلاس‌هایی که موتورهای Java scripting را تعریف می‌کنند و یک چارچوب برای استفاده از آن واسط‌ها و کلاس‌ها در کد جاوا، تشکیل شده‌است. سوءاستفاده از *javax.script* API، به یک مهاجم اجازه می‌دهد تا کد دلخواه را روی سیستم هدف اجرا کند (IDS00-J. داده‌های غیر قابل اعتماد که از یک مرز اعتماد منتقل می‌شوند، پاک‌سازی کنید).

1.7.2 نمونه‌کد ناسازگار

این مثال، از ورودی کاربر غیرقابل اعتماد برای یک عبارت JavaScript استفاده می‌کند، که مسئول پرینت ورودی است:

```
private static void evalScript(String firstName)
      throws ScriptException
      {
            ScriptEngineManager manager = new ScriptEngineManager();
            ScriptEngine engine = manager.getEngineByName("javascript");
            engine.eval("print('"+ firstName + "')");
      }
```

یک مهاجم می‌تواند آرگومانی را، که به‌طور خاصی دستکاری شده‌است، در تلاشی برای تزریق JavaScript مخرب، وارد کند. این مثال، یک رشته‌ی مخرب را نشان می‌دهد که حاوی کد JavaScript است که می‌تواند یک فایل بسازد یا روی یک فایل موجود بر یک سیستم آسیب‌پذیر، بنویسد.

```
dummy\');
var bw = new JavaImporter(java.io.BufferedWriter);
var fw = new JavaImporter(java.io.FileWriter);
with(fw) with(bw)
{
      bwr = new BufferedWriter(new FileWriter(\"config.cfg\"));
      bwr.write(\"some text\"); bwr.close();
} // ;
```

در این مثال، اسکریپت، "*dummy*" را پرینت می‌کند و سپس، "*some text*" را روی یک فایل پیکربندی، به نام config.cfg، می‌نویسد. یک سوءاستفاده‌ی واقعی می‌تواند کد دلخواهی را اجرا کند.

۲.۷.۲ راه‌حل سازگار (لیست‌سفید)

بهترین دفاع در برابر آسیب‌پذیری‌های تزریق کد، جلوگیری از گنجاندن ورودی قابل اجرای کاربر در کد است. ورودی کاربر مورد استفاده در کد پویا باید پاک‌سازی شود تا فقط حاوی کاراکترهای مجاز و موجود در لیست‌سفید باشد. بهترین زمان پاک‌سازی، بلافاصله پس از ورود داده و با استفاده از روش‌هایی از انتزاع داده‌ی مورد استفاده برای ذخیره و پردازش داده‌ها است (IDS00-J. داده‌های غیر قابل اعتماد را که از یک مرز اعتماد منتقل می‌شوند، پاک‌سازی کنید). اگر کاراکترهای خاص باید در نام مجاز باشند، لازم است پیش از مقایسه با فرم‌های معادل برای هدف اعتبارسنجی ورودی، نرمال‌سازی شوند. این راه‌حل، از لیست‌سفید برای جلوگیری از تفسیر ورودی غیرپاک‌سازی‌شده توسط موتور اسکریپت، استفاده می‌کند.

```
private static void evalScript(String firstName)
      throws ScriptException
      {
            // Allow only alphanumeric and underscore chars in firstName
            // (modify if firstName may also include special characters)
            if (!firstName.matches("[\\w]*"))
            {
                  // String does not match whitelisted characters
                  throw new IllegalArgumentException();
            }
            ScriptEngineManager manager = new ScriptEngineManager();
            ScriptEngine engine = manager.getEngineByName("javascript");
            engine.eval("print('"+ firstName + "')");
      }
```

۳.۷.۲ راه‌حل سازگار (جعبه شنی امن)

متد جایگزین، ساخت یک جعبه شنی امن با استفاده از یک مدیریت امنیت است. برنامه باید جلوی اسکریپت را بگیرد تا دستورات دلخواه، مانند پرس‌وجوی[۵۱] سیستم فایل محلی، را اجرا نکند. دو آرگومان *doPrivileged()* را می‌توان برای کاهش امتیازات، هنگامی‌که برنامه باید اما موتور اسکریپت نباید با

[۵۱] Querying

امتیازات بالاتر کار کنند، استفاده نمود. *RestrictedAccessControlContext*، مجوزهای اعطاشده را در فایل سیاست پیش‌فرض برای دامنه‌ی حفاظتی تازه ساخته‌شده، کم می‌کند. مجوزهای موثر، تقاطع دامنه‌ی حفاظتی تازه ساخته‌شده و سیاست امنیت کل سیستم است. این راه‌حل، استفاده از *AccessControlContext* را در نوع دو آرگومانی *doPrivileged()*نشان می‌دهد. برای امنیت بیشتر، این متد می‌تواند با لیست‌سفید ترکیب شود.

```
class ACC
{
	private static class RestrictedAccessControlContext
	{
		private static final AccessControlContext INSTANCE;
		static
		{
			INSTANCE=new AccessControlContext(new ProtectionDomain[]
			{
				new ProtectionDomain(null, null) // No permissions
			});
		}
	}
	private static void evalScript(final String firstName)
	throws ScriptException
	{
		ScriptEngineManager manager = new ScriptEngineManager();
		final ScriptEngine engine =manager.getEngineByName("javascript");
		//Restrict permission using the two-argument form of doPrivileged()
		try {
			AccessController.doPrivileged(
				new PrivilegedExceptionAction<Object>()
			{
				public Object run() throws ScriptException
					{
						engine.eval("print('"+firstName + "')");
							return null;
					}
			},
			// From nested class
			RestrictedAccessControlContext.INSTANCE);
		}
		catch (PrivilegedActionException pae)
		{
			// Handle error
		}
	}
}
```

۴,۷,۲ کاربرد

شکست در جلوگیری از تزریق کد، می‌تواند منجر به اجرای کد دلخواه شود.

۸,۲ از تزریق *XPath* جلوگیری کنید

از زبان XML[۵۲] می‌توان برای ذخیره‌سازی داده‌ها، به‌صورتی مشابه با پایگاه داده‌ی رابطه‌ای استفاده نمود. داده بارها از چنین مستند XML با استفاده از Xpaths بازیابی می‌شود. *تزریق Xpath* می‌تواند هنگامی رخ دهد که داده‌ی فراهم‌شده برای یک روتین بازیابی Xpath، بدون پاک‌سازی مناسب، برای بازیابی داده‌ها از یک مستند XML استفاده شود. این حمله، مشابه تزریق SQL یا XML است (IDS00-J. داده‌های غیرقابل را اعتماد که از یک مرز اعتماد منتقل می‌شوند، پاک‌سازی کنید). یک مهاجم می‌تواند یک ساختار SQL یا XML مجاز را در فیلدهای پرس‌وجوی مورد استفاده، وارد کند. در حملات رایج، فیلد شرطی پرس‌وجو، یک حشو[۵۳] را حل می‌کند. در غیر این‌صورت، امکان دسترسی به اطلاعات دارای حق ویژه[۵۴] را به مهاجم اعطا می‌کند.

۱,۸,۲ نمونه‌ی تزریق مسیر XML

شمای XML زیر را در نظر بگیرید.

```
<users>
	<user>
		<username>Utah</username>
		<password>e90205372a3b89e2</password>
	</user>
	<user>
		<username>Bohdi</username>
		<password>6c16b22029df4ec6</password>
	</user>
	<user>
		<username>Busey</username>
		<password>ad39b3c2a4dabc98</password>
	</user>
</users>
```

[۵۲] Extensible Markup Language

[۵۳] Tautology

[۵۴] Privileged

رمزهای عبور، درهم‌سازی شده‌اند. درهم‌سازهای *MD5*، به‌دلیل گویایی، ذکر شده‌اند؛ در عمل، باید از الگوریتم‌های امن‌تری مانند *SHA-256* استفاده نمود. ممکن است کد غیرقابل اعتماد، برای بازیابی جزئیات کاربر، از این فایل با عبارت Xpath، که به‌صورت پویا از ورودی کاربر ساخته شده‌است، استفاده کند.

```
//users/user[username/text()='&LOGIN&' and
password/text()='&PASSWORD&' ]
```

اگر یک مهاجم بداند که *Utah* یک نام کاربری مجاز است، می‌تواند یک ورودی مانند زیر بسازد:

```
Utah' or '1'='1
```

که منجر به رشته‌ی پرس‌وجوی زیر می‌شود:

```
//users/user[username/text()='Utah' or '1'='1'
and password/text()='xxxx']
```

از آنجایی‌که '1'='1' همواره درست است، رمز عبور هرگز اعتبارسنجی نمی‌شود. در نتیجه، مهاجم به‌صورت نامناسب، به‌عنوان کاربر *Utah* و بدون دانستن رمز عبور *Utah*، احراز اصالت می‌شود.

۲.۸.۲ نمونه‌کد ناسازگار

این مثال، یک نام کاربری و رمز عبور را از کاربر می‌گیرد و از آنها برای ساخت رشته‌ی پرس‌وجو استفاده می‌کند. رمز عبور، به‌عنوان آرایه‌ی کاراکتری فرستاده شده و سپس، درهم‌سازی می‌شود. این مثال، مستعد حمله‌ای که پیش از این توصیف شد، است. اگر رشته‌ی حمله‌ای که پیش از این توصیف شد، به *evaluate()* داده‌شود، فراخوانی، متد گره‌ی مرتبط را در فایل XML برمی‌گرداند و منجر به این می‌شود که متد *doLogin()*، مقدار *true* را برگرداند و هر صدور مجوزی را دور بزند.

```
private boolean doLogin(String userName, char[] password)
throws ParserConfigurationException, SAXException,
IOException, XPathExpressionException
{
	DocumentBuilderFactory domFactory = DocumentBuilderFactory.newInstance();
	domFactory.setNamespaceAware(true);
	DocumentBuilder builder = domFactory.newDocumentBuilder();
	Document doc = builder.parse("users.xml");
	String pwd = hashPassword( password);
	XPathFactory factory = XPathFactory.newInstance();
	XPath xpath = factory.newXPath();
	XPathExpression expr = xpath.compile("//users/user[username/text()='" +
		userName + "' and password/text()='" + pwd + "' ]");
	Object result = expr.evaluate(doc, XPathConstants.NODESET);
	NodeList nodes = (NodeList) result;
	// Print first names to the console
	for (int i = 0; i < nodes.getLength(); i++)
	{
		Node node = nodes.item(i).getChildNodes().item(1).
			getChildNodes().item(0);
		System.out.println("Authenticated: " + node.getNodeValue());
	}
	return (nodes.getLength() >= 1);
}
```

۳,۸,۲ راه‌حل سازگار (XQuery)

می‌توان با اتخاذ پدافندهایی مشابه با جلوگیری از تزریق SQL، از تزریق Xpath جلوگیری نمود.

- با تمام ورودی‌های کاربر، به‌عنوان غیرقابل اعتماد برخورد کنید و پاک‌سازی مناسب را انجام دهید.
- هنگام پاک‌سازی ورودی کاربر، صحت نوع داده، طول، فرمت و محتوا را وارسی کنید. به‌عنوان نمونه، از یک عبارت باقاعده[۵۵]، که تگ‌های XML و کاراکترهای خاص در ورودی کاربر را بررسی می‌کند، استفاده نمایید. این عمل مرتبط با پاک‌سازی است.
- در برنامه‌های کاربردی مشتری-خدمتگزار، اعتبارسنجی را در هر دو طرف مشتری و سرور انجام دهید.

[۵۵] Regular expression

- برنامه‌هایی که ورودی کاربر را فراهم و منتشر می‌کنند یا آن را می‌پذیرند، به‌طور گسترده آزمایش نمایید.

یک روش موثر برای جلوگیری از مسائل مرتبط با تزریق SQL، پارامتری کردن است. پارامتری کردن از آن جهت که داده‌های مختص کاربر به یک API، به‌گونه‌ای به‌عنوان پارامتر داده‌شوند که داده هرگز به‌عنوان محتوای اجرایی تفسیر نشود، اطمینان حاصل می‌کند. متاسفانه، اکنون Java SE فاقد واسطی مشابه با جست‌وجوهای Xpath است. اگرچه، Xpath، در مقایسه با پارامتری کردن SQL، می‌تواند با استفاده از واسطی مانند Xquery، که از مشخص کردن یک عبارت پرس‌وجو در یک فایل جداگانه‌ی مهیاشده در زمان اجرا پشتیبانی می‌کند، شبیه‌سازی شود.

فایل ورودی: login.xq

```
declare variable $userName as xs:string external;
declare variable $password as xs:string external;
//users/user[@userName=$userName and @password=$password]
```

این راه‌حل، با استفاده از خواندن فایل با فرمت مورد نیاز و سپس افزودن مقادیر برای نام کاربری و رمز عبور در یک Map، از پرس‌وجوی مشخص‌شده در یک فایل متنی استفاده می‌کند. کتابخانه‌ی Xquery، پرس‌وجوی XML را برای این ورودی‌ها می‌سازد. با استفاده از این متد، داده‌های مشخص‌شده در فیلدهای نام کاربری و رمز عبور نمی‌توانند به‌عنوان محتوای اجرایی در زمان اجرا تفسیر شوند.

```
private boolean doLogin(String userName, String pwd)
throws ParserConfigurationException, SAXException,
IOException, XPathExpressionException
{
	DocumentBuilderFactory domFactory = DocumentBuilderFactory.newInstance();
	domFactory.setNamespaceAware(true);
	DocumentBuilder builder = domFactory.newDocumentBuilder();
	Document doc = builder.parse("users.xml");
	XQuery xquery = new XQueryFactory().createXQuery(new File("login.xq"));
	Map queryVars = new HashMap();
	queryVars.put("userName", userName);
	queryVars.put("password", pwd);
	NodeList nodes = xquery.execute(doc, null, queryVars).toNodes();
	// Print first names to the console
	for (int i = 0; i < nodes.getLength(); i++)
	{
		Node node = nodes.item(i).getChildNodes().item(1).
		getChildNodes().item(0);
		System.out.println(node.getNodeValue());
	}
	return (nodes.getLength() >= 1);
}
```

۴,۸,۲ کاربرد

ممکن است شکست در اعتبارسنجی ورودی کاربر، منجر به افشا و اجرای کد غیرممتاز[۵۶] شود. جلوگیری از تزریق XPath، نیازمند حذف (که نهی شده‌است) یا رهایی مناسب ویژگی‌های زیر است.

- " = ' / < >، برای جلوگیری از تزریق پارامتر مستقیم.
- پرس‌وجوهای Xpath نباید حاوی هیچ فراکاراکتری (مانند // ? * = ' یا مشابه آنها) باشند.
- توسعه‌ی XSLT نباید حاوی هیچ ورودی کاربری باشد. اگر چنین است، باید وجود فایل به‌صورت گسترده‌ای آزمایش شده و اطمینان حاصل شود که فایل‌ها درون مرزهای تعیین‌شده توسط سیاست امنیت جاوا ۲[۵۷] هستند.

[۵۶] Unprivileged

[۵۷] Java 2 Security Policy

۹,۲ از تزریق LDAP جلوگیری کنید

پروتکل LDAP[۵۸]، به یک برنامه‌ی کاربردی اجازه می‌دهد تا از راه دور عملیاتی، مانند جست‌وجو و اصلاح رکوردها در دایرکتوری‌ها، را انجام دهد. تزریق LDAP، ناشی از پاک‌سازی و اعتبارسنجی ناکافی ورودی است و به کاربران مخرب اجازه می‌دهد اطلاعات محدو شده را با استفاده از سرویس دایرکتوری، جمع کنند[۵۹].

می‌توان از یک لیست‌سفید برای محدود کردن ورودی به لیستی از کاراکترهای مجاز استفاده کرد. کاراکترها و دنبال‌های کاراکتری که باید از لیست‌های سفید حذف شوند، شامل فراکاراکترهای JNDI[۶۰] و کاراکترهای خاص LDAP هستند، در جدول ۲-۱ آورده شده‌اند.

جدول ۲-۱: کاراکترها و دنباله‌هایی که باید از لیست سفید حذف شوند

کاراکتر	نام
‘ و “	کوتیشن و دابل‌کوتیشن
/ و \	اِسلَش و بَک‌اِسلَش
\\	دابل‌اِسلَش
space	فضای خالی (اِسپِیس) در ابتدا و انتهای رشته
#	کاراکتر هَش در ابتدای رشته
< و >	براکت‌های زاویه‌دار
, و ;	ویرگول (کاما) و نقطه‌ویرگول (سِمی‌کُلون)
+ و *	عملگرهای جمع و ضرب
(و)	پرانتزهای باز و بسته
\u0000	یونیکُد کاراکتر NULL

[۵۸] Lightweight Directory Access Protocol

[۵۹] Glean

[۶۰] Java Naming and Directory Interface

۱,۹,۲ مثال تزریق LDAP

یک فایل LDIF[۶۱] را در نظر بگیرید که حاوی رکوردهایی به فرمت زیر است:

```
dn: dc=example,dc=com
objectclass: dcobject
objectClass: organization
o: Some Name
dc: example

dn: ou=People,dc=example,dc=com
ou: People
objectClass: dcobject
objectClass: organizationalUnit
dc: example

dn: cn=Manager,ou=People,dc=example,dc=com
cn: Manager
sn: John Watson
# Several objectClass definitions here (omitted)
userPassword: secret1
mail: john@holmesassociates.com

dn: cn=Senior Manager,ou=People,dc=example,dc=com
cn: Senior Manager
sn: Sherlock Holmes
# Several objectClass definitions here (omitted)
userPassword: secret2
mail: sherlock@holmesassociates.com
```

یک جست‌وجو برای یک نام کاربری و رمز عبور مجاز، معمولاً فرم زیر را به خود می‌گیرد:

```
(&(sn=<USERSN>)(userPassword=<USERPASSWORD>))
```

با این‌حال، یک مهاجم می‌تواند احراز اصالت را با استفاده از *S برای فیلد *USERSN*، و * برای فیلد *USERPASSWORD*، دور بزند. این ورودی، به هر رکوردی که فیلد *USERSN* آن با *S* شروع می‌شود، دست می‌یابد.

[۶۱] LDAP Data Interchange Format

روتین احراز اصالتی که امکان تزریق LDAP را فراهم می‌کند، اجازه‌ی ورود به سیستم را به کاربران غیرمجاز می‌دهد. به‌طور مشابه، یک روتین جست‌وجو، به یک مهاجم اجازه می‌دهد تا بخشی یا تمام داده‌های موجود در دایرکتوری را کشف کند.

۲.۹.۲ نمونه‌کد ناسازگار

این مثال، به یک فراخواننده‌ی متد *searchRecord()* اجازه می‌دهد تا با استفاده از پروتکل LDAP، رکوردی را در دایرکتوری جست‌وجو کند. فیلتر رشته، برای فیلتر کردن مجموعه نتایج ورودی‌هایی که با یک نام کاربری و رمز عبور فراهم‌شده توسط فراخواننده مطابقت دارند، استفاده می‌شود.

```
// String userSN = "S*"; // Invalid
// String userPassword = "*"; // Invalid
public class LDAPInjection {
	private void searchRecord(String userSN, String userPassword)
	throws NamingException {
		Hashtable<String, String> env = new Hashtable<String, String>();
		env.put(Context.INITIAL_CONTEXT_FACTORY,
			"com.sun.jndi.ldap.LdapCtxFactory");
		try {
			DirContext dctx = new InitialDirContext(env);
			SearchControls sc = new SearchControls();
			String[] attributeFilter = {"cn", "mail"};
			sc.setReturningAttributes(attributeFilter);
			sc.setSearchScope(SearchControls.SUBTREE_SCOPE);
			String base = "dc=example,dc=com";
			// The following resolves to (&(sn=S*)(userPassword=*))
			String filter = "(&(sn=" + userSN + ")(userPassword=" +
				userPassword + "))";
			NamingEnumeration<?> results =dctx.search(base, filter, sc);
			while (results.hasMore()) {
				SearchResult sr = (SearchResult) results.next();
				Attributes attrs = (Attributes) sr.getAttributes();
				Attribute attr = (Attribute) attrs.get("cn");
				System.out.println(attr);
				attr = (Attribute) attrs.get("mail");
				System.out.println(attr);
			}
			dctx.close();
		} catch (NamingException e) {
			// Forward to handler
		}
	}
}
```

هنگامی‌که کاربر مخربی داده‌هایی را که به‌صورت خاصی دستکاری شده‌اند وارد می‌کند، شِمای احراز اصالت ابتدایی، در محدود کردن خروجی پرس‌وجوی جست‌وجو به اطلاعاتی که کاربر امتیاز دسترسی به آنها را دارد، شکست می‌خورد.

۳.۹.۲ راه‌حل سازگار

این راه‌حل، از یک لیست‌سفید برای پاک‌سازی ورودی کاربر استفاده می‌کند تا رشته‌ی فیلتر، تنها حاوی کاراکترهای مجاز باشد. در این کد، *userSN* می‌تواند تنها حاوی حروف و فاصله‌ها باشد. این در حالی است که رمز عبور می‌تواند تنها حاوی کاراکترهای حروف و رقم[۶۲] باشد.

```
// String userSN = "Sherlock Holmes"; // Valid
// String userPassword = "secret2"; // Valid
// ... beginning of LDAPInjection.searchRecord() ...
sc.setSearchScope(SearchControls.SUBTREE_SCOPE);
String base = "dc=example,dc=com";
if (!userSN.matches("[\\w\\s]*") || !userPassword.matches("[\\w]*"))
{
      throw new IllegalArgumentException("Invalid input");
}
String filter = "(&(sn = " + userSN + ")(userPassword=" + userPassword + "))";
// ... remainder of LDAPInjection.searchRecord() ...
```

وقتی یک فیلد پایگاه داده، مانند یک رمز عبور، باید حاوی کاراکترهای خاص باشد، حصول اطمینان از این‌که داده‌های معتبر، به‌صورت پاک‌سازی‌شده، در پایگاه داده ذخیره شده‌اند و همچنین هر ورودی کاربر پیش از وقوع اعتبارسنجی یا مقایسه، نرمال‌سازی شده‌است، امری حیاتی است. استفاده از کاراکترهایی که در غیاب یک نرمال‌سازی گسترده و روتین مبتنی بر لیست‌سفید، از معانی خاصی در JNDI و LDAP برخوردار هستند، توصیه نمی‌شود. کاراکترهای خاص پیش از این‌که به عبارت لیست‌سفیدی که ورودی نسبت به آن اعتبارسنجی می‌شود اضافه شوند، باید به مقادیر پاک‌سازی‌شده و امن تبدیل گردند. به‌طور مشابه، نرمال‌سازی ورودی کاربر باید پیش از گام اعتبارسنجی رخ دهد.

[۶۲] Alphanumeric

۴,۹,۲ کاربرد

شکست در پاک‌سازی ورودی غیرقابل اعتماد می‌تواند منجر به افشای اطلاعات و افزایش امتیازات شود.

۱۰,۲ از متد *clone()* برای کپی کردن پارامترهای متد غیرقابل اعتماد استفاده نکنید

ساخت کپی‌های تدافعی از پارامترهای متد قابل تغییر، در برابر انواع آسیب‌پذیری‌های امنیتی کاهش می‌یابد(OBJ06-J. ورودی‌های تغییرپذیر و مولفه‌های داخلی تغییرپذیر را به‌صورت تدافعی کپی کنید).

با این‌وجود، استفاده‌ی نامناسب از متد *clone()* می‌تواند به یک مهاجم اجازه دهد تا با استفاده از آرگومان‌هایی که طبیعی به‌نظر می‌رسند اما در نهایت مقادیر غیرمنتظره را برمی‌گردانند، از آسیب‌پذیری‌ها سوءاستفاده کند. چنین اشیایی، در نهایت ممکن است بررسی‌های امنیتی و اعتبارسنجی را دور بزنند. هنگامی‌که چنین کلاسی ممکن است به‌عنوان آرگومانی به یک متد ارسال شود، با آرگومان به‌عنوان غیرقابل اعتماد برخورد کنید و از متد clone() مهیا شده توسط کلاس، استفاده ننمایید. همچنین، از متد *clone()* کلاس‌های غیرنهایی برای ایجاد کپی‌های تدافعی استفاده نکنید.

۱,۱۰,۲ نمونه‌کد ناسازگار

این مثال، یک متد *validateValue()* تعریف می‌کند که یک مقدار زمانی را اعتبارسنجی می‌نماید:

```
private Boolean validateValue(long time)
{
      // Perform validation
      return true; // If the time is valid
}
private void storeDateInDB(java.util.Date date)
throws SQLException
{
      final java.util.Date copy = (java.util.Date)date.clone();
      if (validateValue(copy.getTime()))
      {
            Connection con = DriverManager.getConnection(
                  "jdbc:microsoft:sqlserver://<HOST>:1433","<UID>", "<PWD>");
            PreparedStatement pstmt = con.prepareStatement("UPDATE ACCESSDB SET
                  TIME = ?");
            pstmt.setLong(1, copy.getTime());
            // ...
      }
}
```

متد *storeDateInDB()*، یک آرگومان داده‌ای غیرقابل اعتماد را می‌پذیرد و تلاش می‌کند تا با استفاده از متد *clone()*، یک کپی تدافعی ایجاد نماید. این کار به مهاجم اجازه می‌دهد تا با ساخت کلاس *date*

مخرب که از *Date*، توسعه یافته‌است، کنترل برنامه را به دست گیرد. اگر کد مهاجم با امتیازات مشابه، توسط *store-DateInDB()* اجرا شود، مهاجم به سادگی کد مخرب را درون متد *clone()* آنها تعبیه می‌کند:

```
class MaliciousDate extends java.util.Date
{
	@Override
	public MaliciousDate clone()
	{
		// malicious code goes here
	}
}
```

با این‌وجود، اگر مهاجم فقط بتواند یک تاریخ مخرب را با امتیازات کاهش‌یافته فراهم کند، می‌تواند اعتبارسنجی را دور بزند، اما هنوز باقی‌مانده‌ی برنامه را دچار پریشانی می‌کند. این مثال را در نظر بگیرید:

```
public class MaliciousDate extends java.util.Date
{
	private static int count = 0;
	@Override
	public long getTime()
	{
		java.util.Date d = new java.util.Date();
		return (count++ == 1) ? d.getTime() : d.getTime() - 1000;
	}
}
```

اولین باری که *getTime()* فراوانی می‌شود، تاریخ مخرب، یک شی تاریخ بی‌خطر به‌نظر می‌رسد. این امر به آن اجازه می‌دهد تا اعتبارسنجی در روش *storeDateInDB()* را دور بزند. اگرچه، زمانی که واقعاً در پایگاه داده ذخیره می‌شود، نادرست خواهد بود.

۲.۱۰.۲ راه‌حل سازگار

این راه‌حل، از به‌کارگیری متد *clone()* اجتناب می‌کند. در عوض، یک شی *java.util.Date* جدید می‌سازد که در نهایت، به‌منظور بررسی‌های کنترل دسترسی و اضافه نمودن به پایگاه داده، استفاده می‌شود.

```
private void storeDateInDB(java.util.Date date)
throws SQLException
{
	final java.util.Date copy = new java.util.Date(date.getTime());
	if (validateValue(copy.getTime()))
	{
		Connection con = DriverManager.getConnection(
			"jdbc:microsoft:sqlserver://<HOST>:1433","<UID>", "<PWD>");
		PreparedStatement pstmt =
			con.prepareStatement("UPDATE ACCESSDB SET TIME = ?");
		pstmt.setLong(1, copy.getTime());
		// ...
	}
}
```

۳،۱۰،۲نمونه‌کد ناسازگار (CVE-2012-0507)

این مثال، یک سازنده[۶۳] از کلاس هسته‌ی جاوا *AtomicReferenceArray* را، که در به‌روزرسانی دوم Java 1.7.0 ارائه شد، نشان می‌دهد:

```
public AtomicReferenceArray(E[] array)
{
	// Visibility guaranteed by final field guarantees
	this.array = array.clone();
}
```

این کد، توسط سوءاستفاده‌ی *Flashback*، که ۶۰۰۰۰۰ کامپیوتر مکینتاش را در سال ۲۰۱۲ آلود کرد، فراخوانی شد.

۴،۱۰،۲راه‌حل سازگار (CVE-2012-0507)

سازنده، در به‌روزرسانی سوم Java 1.7.0، برای استفاده از متد *Arrays.copyOf()* به جای متد *clone()*، به‌صورت زیر اصلاح شد:

```
public AtomicReferenceArray(E[] array)
{
	// Visibility guaranteed by final field guarantees
	this.array = Arrays.copyOf(array, array.length, Object[].class);
}
```

[۶۳] Constructor

۵,۱۰,۲ کاربرد

استفاده از متد *clone()* برای کپی کردن آرگومان‌های غیرقابل اعتماد، به مهاجمان فرصت اجرای کد دلخواه را می‌دهد.

۱۱,۲ از *Object.equals()* برای مقایسه‌ی کلیدهای رمزنگاری استفاده نکنید

متد *method java.lang.Object.equals()*، به‌صورت پیش‌فرض، قادر به مقایسه‌ی اشیای ترکیبی، مانند کلیدهای رمزنگاری نیست. بیشتر کلاس‌های *Key*، در فراهم نمودن یک پیاده‌سازی *equals()*، که *Object.equals()* را لغو کند[۶۴]، شکست می‌خورند. در چنین مواردی، مولفه‌های شی ترکیبی باید به‌منظور اطمینان یافتن از درستی، به‌طور جداگانه مقایسه شوند.

۱,۱۱,۲ نمونه کد ناسازگار

این مثال ، دو کلید را با استفاده از *equals()* مقایسه می‌کند. ممکن است کلیدها، حتی اگر مقدار یکسانی را ارائه کنند، نامساوی درنظر گرفته شوند.

```
private static boolean keysEqual(Key key1, Key key2)
{
	if (key1.equals(key2))
	{
		return true;
	}
	return false;
}
```

۲,۱۱,۲ راه‌حل سازگار

این راه‌حل، از *equals()* به‌عنوان اولین آزمایش استفاده می‌کند. سپس، نسخه‌ی رمزنگاری‌شده‌ی کلیدها را، به‌منظور تسهیل رفتار مستقل از عرضه‌کنند، مقایسه می‌نماید. در نهایت، بررسی می‌کند که آیا یک *RSAPrivateKey* و یک *RSAPrivateCrtKey*، نشان‌دهنده‌ی کلیدهای خصوصی معادل هستند یا خیر.

[۶۴] Overrides

```
private static boolean keysEqual(Key key1, Key key2)
{
	if (key1.equals(key2))
	{
		return true;
	}
	if (Arrays.equals(key1.getEncoded(), key2.getEncoded()))
	{
		return true;
	}
	// More code for different types of keys here. For example, the following
	// code can check whether an RSAPrivateKey and an RSAPrivateCrtKey are
	// equal
	if ((key1 instanceof RSAPrivateKey) && (key2 instanceof RSAPrivateKey))
	{
		if ((((RSAKey) key1).getModulus().equals(
		((RSAKey) key2).getModulus())) &&
		(((RSAPrivateKey) key1).getPrivateExponent().equals(
		((RSAPrivateKey) key2).getPrivateExponent())))
		{
			return true;
		}
	}
	return false;
}
```

۳,۱۱,۲تشخیص خودکار

ممکن است استفاده از *Object.equals()* کلیدهای رمزنگاری برای مقایسه، به نتایج غیرمنتظره برسد.

۱۲,۲ از الگوریتم‌های رمزنگاری غیرامن یا ضعیف استفاده نکنید

لازم است برنامه‌های کاربردی با امنیت شدید، از اصول رمزنگاری غیرامن یا ضعیف دوری کنند. ظرفیت محاسباتی کامپیوترهای جدید، امکان دور زدن چنین رمزنگاری‌هایی را از طریق حملات کورکورانه فراهم می‌کند. به‌عنوان نمونه، الگوریتم رمزنگاری DES[۶۵]، به‌شدت ناامن در نظر گرفته می‌شود؛ پیام‌هایی که با استفاده از DES رمزنگاری می‌شوند، با متد کورکورانه، تنها طی یک روز و توسط ماشینی مانند Electronic Frontier Foundation's (EFF) Deep Crack، رمزگشایی می‌شوند.

1.12.2 نمونه‌کد ناسازگار

این مثال، یک رشته‌ی ورودی را با استفاده الگوریتم رمزنگاری ضعیف (DES)، رمزنگاری می‌کند:

```
SecretKey key = KeyGenerator.getInstance("DES").generateKey();
Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.ENCRYPT_MODE, key);
// Encode bytes as UTF8; strToBeEncrypted contains
// the input string that is to be encrypted
byte[] encoded = strToBeEncrypted.getBytes("UTF8");
// Perform encryption
byte[] encrypted = cipher.doFinal(encoded);
```

۲,۱۲,۲راه‌حل سازگار

این راه‌حل، از الگوریتم امن‌تر AES[۶۶] برای انجام رمزنگاری استفاده می‌کند.

```
Cipher cipher = Cipher.getInstance("AES");
KeyGenerator kgen = KeyGenerator.getInstance("AES");
kgen.init(128); // 192 and 256 bits may be unavailable
SecretKey skey = kgen.generateKey();
byte[] raw = skey.getEncoded();
SecretKeySpec skeySpec = new SecretKeySpec(raw, "AES");
cipher.init(Cipher.ENCRYPT_MODE, skeySpec);
// Encode bytes as UTF8; strToBeEncrypted contains the
// input string that is to be encrypted
byte[] encoded = strToBeEncrypted.getBytes("UTF8");
// Perform encryption
byte[] encrypted = cipher.doFinal(encoded);
```

[۶۵] Data Encryption Standard

[۶۶] Advanced Encryption Standard

۳,۱۲,۲ کاربرد

استفاده از الگوریتم‌های رمزنگاری ناامن ریاضیاتی و محاسباتی، می‌تواند منجر به افشای اطلاعات حساس شود. الگوریتم‌های رمزنگاری ضعیف را می‌توان در Java SE 7 غیرفعال نمود.

از الگوریتم‌های رمزنگاری ضعیف می‌توان در سناریوهایی که به طور خاص به دنبال یک رمز[۶۷] قابل شکستن هستند، استفاده نمود. به‌عنوان نمونه، هنگامی‌که هدف رمزنگاری، محافظت از مردم در برابر اطلاعات است (نه محافظت از اطلاعات در برابر مردم)، از رمز ROT13 برای تابلوهای اعلانات و وب‌سایت‌ها استفاده می‌شود.

[۶۷] Cipher

۱۳,۲ رمزهای عبور را با استفاده از یک تابع درهم‌سازی، ذخیره کنید

برنامه‌هایی که رمزهای عبور را به‌صورت متنی ساده (داده‌ی متنی بدون رمزنگاری) ذخیره می‌کنند، خطر افشای آن رمزهای عبور را به طرق مختلف ایجاد می‌نمایند. اگرچه برنامه‌ها، رمزهای عبور را از کاربران به‌صورت متن ساده دریافت می‌کنند، اما باید اطمینان یابند که این رمزهای عبور، به‌صورت متن ساده ذخیره نمی‌شوند.

یک متد قابل پذیرش برای محدود نمودن افشای رمزهای عبور، استفاده از *توابع درهم‌سازی* است، که به برنامه‌ها اجازه می‌دهد تا به‌صورت غیرمستقیم، یک رمز عبور ورودی را با رشته‌ی رمز عبور اصلی (بدون ذخیره‌سازی به‌صورت متن ساده یا نسخه‌ی قابل رمزگشایی رمز عبور) مقایسه کنند. این متد، افشای رمز عبور را بدون ایجاد نقطه‌ضعف خاصی، کاهش می‌دهد.

1.13.2 توابع رمزنگاری درهم‌سازی

مقدار تولیدشده توسط یک تابع درهم‌سازی، *مقدار درهم‌سازی* یا *خلاصه‌پیام*[۶۸] نامیده می‌شود. توابع درهم‌سازی، توابعی هستند که از لحاظ محاسباتی امکان‌پذیرند[۶۹]، اما معکوس آنها از لحاظ محاسباتی امکان‌پذیر نیست. در عمل، یک رمز عبور می‌تواند به یک مقدار درهم‌سازی، رمزنگاری شود، اما رمزگشایی امکان‌ناپذیر باقی می‌ماند. یکسانی رمزهای عبور می‌تواند از طریق یکسانی مقادیر درهم‌سازی آنها، آزمایش شود.

یک کار خوب در این زمینه، همواره افزودن *نمک*[۷۰] به رمز عبور در حال درهم‌سازی است. نمک، یک تکه داده‌ی منحصر به‌فرد (و معمولاً ترتیبی) یا به‌طور تصادفی تولیدشده است که با مقدار درهم‌سازی ذخیره می‌شود. استفاده از یک نمک، به مقادیر درهم‌سازی در جلوگیری از حملات کورکورانه کمک می‌کند، به شرطی‌که نمک به اندازه‌ی کافی بلند باشد تا آنتروپی[۷۱] کافی تولید نماید (مقادیر کوتاهتر نمک، نمی‌توانند به‌میزان قابل ملاحظه‌ای، از سرعت حملات کورکورانه بکاهند). هر رمز عبور باید نمک مخصوص

---

[۶۸] Message digest
[۶۹] Feasible
[۷۰] Salt
[۷۱] Entropy

و مرتبط به خود را داشته‌باشد. اگر یک نمک برای بیشتر از یک رمز عبور استفاده شود، دو کاربر قادر به مشاهده‌ی این‌که آیا رمزهای عبور مشابه هستند یا خیر، خواهند بود.

انتخاب تابع درهم‌سازی و طول نمک، توازنی بین امنیت و کارایی ارائه می‌کند. افزایش تلاش‌های لازم برای حملات کورکورانه‌ی موثر، با انتخاب یک تابع درهم‌سازی قوی‌تر می‌تواند زمان لازم برای اعتبارسنجی یک رمز عبور را نیز افزایش دهد. افزایش طول نمک، انجام حملات کورکورانه را دشوارتر می‌سازد، اما نیازمند فضای ذخیره‌سازی اضافی است.

کلاس *MessageDigest* جاوا، پیاده‌سازی‌هایی از توابع رمزنگاری درهم‌سازی گوناگون را ارائه می‌کند. از توابع تدافعی مانند *MD5*[۷۲] دوری کنید. توابع درهم‌سازی مانند *SHA-1*[۷۳] و *SHA-2*، توسط آژانس امنیت ملی[۷۴] نگهداری می‌شوند، و اکنون به‌عنوان ایمن درنظر گرفته می‌شوند. در عمل، بسیاری از برنامه‌های کاربردی، از *SHA-256* استفاده می‌کنند، زیرا این تابع درهم‌سازی، کارایی معقولی دارد و همچنان امن است.

۲,۱۳,۲ نمونه‌کد ناسازگار

این مثال، رمز عبور ذخیره‌شده در *password.bin* را با استفاده از یک الگوریتم کلید متقارن، رمزنگاری و رمزگشایی می‌کند.

یک مهاجم به‌طور بالقوه، به‌خصوص هنگامی‌که از دانشی در مورد کلید و شمای رمزنگاری مورداستفاده توسط برنامه برخوردار باشد، می‌تواند این فایل را رمزگشایی کند تا رمز عبور را کشف نماید. رمزهای عبور باید حتی از مدیران سیستم و کاربران ممتاز، محافظت شوند. در نتیجه، استفاده از رمزنگاری، فقط تا حدی در کاهش تهدیدات افشا رمز عبور موثر است.

---

[۷۲] Message-Digest Algorithm

[۷۳] Secure Hash Algorithm

[۷۴] National Security Agency

```
public final class Password
{
	private void setPassword(byte[] pass) throws Exception
	{
		// Arbitrary encryption scheme
		bytes[] encrypted = encrypt(pass);
		clearArray(pass);
		// Encrypted password to password.bin
		saveBytes(encrypted,"password.bin");
		clearArray(encrypted);
	}
	boolean checkPassword(byte[] pass) throws Exception
	{
		// Load the encrypted password
		byte[] encrypted = loadBytes("password.bin");
		byte[] decrypted = decrypt(encrypted);
		boolean arraysEqual = Arrays.equal(decrypted, pass);
		clearArray(decrypted);
		clearArray(pass);
		return arraysEqual;
	}
	private void clearArray(byte[] a)
	{
		for (int i = 0; i < a.length; i++)
		{
		a[i] = 0;
		}
	}
}
```

۳,۱۳,۲نمونه‌کد ناسازگار

این مثال، از تابع درهم‌سازی *SHA-256* از طریق کلاس *MessageDigest*، برای مقایسه‌ی مقادیر درهم‌سازی به جای متن ساده، استفاده می‌کند. اما، از *sting* برای ذخیره‌سازی رمز عبور بهره می‌برد:

حتی وقتی مهاجم بداند که برنامه رمزهای عبور را با استفاده از *SHA-256* و یک نمک ۱۲ بایتی ذخیره می‌کند، قادر نخواهد بود که رمز عبور واقعی را از *password.bin* و *salt.bin* بازیابی نماید. اگرچه، این متد مساله‌ی رمزگشایی از مثال کد ناسازگار پیشین را برطرف می‌کند، اما ممکن است این برنامه، سهواً رمزهای عبور را به‌صورت متن ساده در حافظه ذخیره نماید. اشیای *string* جاوا، تغییرناپذیر هستند و می‌توانند کپی

شوند و به‌صورت داخلی، در JVM ذخیره گردند. در نتیجه، جاوا فاقد مکانیزمی برای پاک کردن امن یک رمز عبور، پس از ذخیره‌سازی آن در یک *string* است.

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
public final class Password
{
	private void setPassword(String pass) throws Exception
	{
		byte[] salt = generateSalt(12);
		MessageDigest msgDigest = MessageDigest.getInstance("SHA-256");
		// Encode the string and salt
		byte[] hashVal = msgDigest.digest((pass+salt).getBytes());
		saveBytes(salt, "salt.bin");
		// Save the hash value to password.bin
		saveBytes(hashVal,"password.bin");
	}
	boolean checkPassword(String pass) throws Exception
	{
		byte[] salt = loadBytes("salt.bin");
		MessageDigest msgDigest = MessageDigest.getInstance("SHA-256");
		// Encode the string and salt
		byte[] hashVal1 = msgDigest.digest((pass+salt).getBytes());
		// Load the hash value stored in password.bin
		byte[] hashVal2 = loadBytes("password.bin");
		return Arrays.equals(hashVal1, hashVal2);
	}
	private byte[] generateSalt(int n)
	{
		// Generate a random byte array of length n
	}
}
```

۴,۱۳,۲راه‌حل سازگار

این راه‌حل، با استفاده از یک آرایه‌ی بایتی[۷۵] برای ذخیره‌سازی رمز عبور، به مسائل نمونه‌کد ناسازگار پیشین، اشاره می‌کند:

[۷۵] Byte array

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
public final class Password {
	private void setPassword(byte[] pass) throws Exception {
		byte[] salt = generateSalt(12);
		byte[] input = appendArrays(pass, salt);
		MessageDigest msgDigest = MessageDigest.getInstance("SHA-256");
		// Encode the string and salt
		byte[] hashVal = msgDigest.digest(input);
		clearArray(pass);
		clearArray(input);
		saveBytes(salt, "salt.bin");
		// Save the hash value to password.bin
		saveBytes(hashVal,"password.bin");
		clearArray(salt);
		clearArray(hashVal);
	}
	boolean checkPassword(byte[] pass) throws Exception {
		byte[] salt = loadBytes("salt.bin");
		byte[] input = appendArrays(pass, salt);
		MessageDigest msgDigest = MessageDigest.getInstance("SHA-256");
		// Encode the string and salt
		byte[] hashVal1 = msgDigest.digest(input);
		clearArray(pass);
		clearArray(input);
		// Load the hash value stored in password.bin
		byte[] hashVal2 = loadBytes("password.bin");
		boolean arraysEqual = Arrays.equals(hashVal1, hashVal2);
		clearArray(hashVal1);
		clearArray(hashVal2);
		return arraysEqual;
	}
	private byte[] generateSalt(int n)
	{
		// Generate a random byte array of length n
	}
	private byte[] appendArrays(byte[] a, byte[] b)
	{
		// Return a new array of a[] appended to b[]
	}
	private void clearArray(byte[] a)
	{
		for (int i = 0; i < a.length; i++)
		{
			a[i] = 0;
		}
	}
}
```

در هر دو متد *setPassword()* و *checkPassword()*، نمایش متن ساده‌ی رمز عبور، بلافاصله پس از تبدیل آن به یک مقدار درهم‌سازی، پاک می‌شود. در نتیجه، مهاجم باید سخت‌تر کار کند تا رمز عبور متن ساده را، پس از پاک شدن، بازیابی نماید. با فراهم‌سازی پاک کردن تضمین‌شده، این کار به‌شدت چالش برانگیز شده و احتمالاً، مختص پلتفرم می‌شود. حتی ممکن است به‌دلیل کپی کردن زباله‌روب‌ها (GCes)[۷۶]، صفحه‌بندی[۷۷] پویا، و سایر ویژگی‌های پلتفرمی که زیر سطح زبان جاوا عمل می‌کند، ناممکن شود.

۵,۱۳,۲ کاربرد

رمزهای عبور ذخیره‌شده فاقد یک درهم‌سازی امن، می‌توانند در معرض کاربران مخرب قرار گیرند. عموماً نقض این راهنما، سوءاستفاده‌ی آشکاری را در ارتباط با آنها به دنبال دارد.

ممکن است برنامه‌های کاربردی، مانند مدیر رمز عبور، به بازیابی رمز عبور اصلی برای وارد کردن آن به یک برنامه‌ی کاربردی ثالث، احتیاج داشته‌باشند. این‌کار، با وجود این‌که این راهنما را نقض می‌کند، مجاز است. مدیر رمز عبور، توسط یک کاربر منفرد مورد دسترسی قرار می‌گیرد و همواره، اجازه‌ی کاربر را برای ذخیره‌سازی رمزهای عبور او و نمایش آن رمزهای عبور روی دستورات، در اختیار دارد. در نتیجه، فاکتور محدودکننده برای ایمنی و امنیت، صلاحیت[۷۸] کاربر است نه عملکرد برنامه.

---

[۷۶] Garbage collectors

[۷۷] Paging

[۷۸] Competence

۱۴,۲ اطمینان حاصل کنید که *SecureRandom*، به‌خوبی مقداردهی اولیه شده‌باشد

تولید عدد تصادفی به یک منبع آنتروپی، مانند سیگنال‌ها، دستگاه‌ها، یا ورودی‌های سخت‌افزاری، بستگی دارد (MSC02-J. اعداد تصادفی قوی تولید کنید).

کلاس *java.security.SecureRandom* به‌طور گسترده، برای تولید اعداد تصادفی، که از لحاظ رمزنگاری قوی هستند، استفاده می‌شود. طبق فایل *java.security* عرضه‌شده در پوشه‌ی *Java Runtime Environment's lib/security*:

منبع داده‌ی مقداردهی اولیه‌شده[۷۹] را برای *SecureRandom* انتخاب کنید. به‌صورت پیش‌فرض، تلاش برای استفاده از دستگاه جمع‌آوری‌کننده‌ی آنتروپی دستگاهی که توسط *securerandom.source* مشخص شده‌است، صورت می‌پذیرد. اگر هنگام دست‌یابی به URL، استثنایی رخ دهد، آن‌گاه، از الگوریتم سنتی فعالیت سیستم/ نخ استفاده می‌شود.

در سیستم‌های Solaris و Linux، اگر file:/dev/urandom مشخص شود و وجود داشته‌باشد، یک پیاده‌سازی خاص *SecureRandom*، به‌صورت پیش‌فرض فعال می‌شود. این "*NativePRNG*"، بایت‌های تصادفی را مستقیما از /dev/urandom می‌خواند. در سیستم‌های Windows، URLهای file:/dev/random و file:/dev/urandom، استفاده از عملکرد مقداردهی اولیه‌ی Microsoft CryptoAPI را فعال می‌سازند.

یک مهاجم نباید قادر به تعیین آغازگر اصلی چندین نمونه از اعداد تصادفی باشد. اگر این محدودیت نقض شود، ممکن است تمام ویژگی‌های اعداد تصادفی، به درستی توسط مهاجم پیش‌بینی شوند.

1.14.2 نمونه‌کد ناسازگار

این مثال، یک تولیدکننده‌ی عدد تصادفی را می‌سازد که با بایت‌های دانه‌ی خاصی، مقداردهی اولیه شده‌است.

---

[۷۹] Seeded

```
SecureRandom random = new SecureRandom(
      String.valueOf(new Date().getTime()).getBytes());
```

این سازنده، ثبّاتی از عرضه‌کنندگان امنیت را جست‌وجو می‌کند و اولین عرضه‌کننده‌ای را که از تولید عدد تصادفی امن پشتیبانی می‌نماید، برمی‌گرداند. اگر چنین عرضه‌کننده‌ای وجود نداشت، یک پیش‌فرض، مختص پیاده‌سازی انتخاب می‌شود. علاوه بر این، نقطه‌ی آغازین پیش‌فرض فراهم‌شده توسط سیستم، توسط نقطه‌ی آغازین فراهم‌شده توسط برنامه‌نویس، بازنویسی می‌شود. استفاده از زمان فعلی سیستم به‌عنوان نقطه‌ی آغازین، قابل پیش‌بینی است و می‌تواند منجر به تولید اعداد تصادفی با آنتروپی ناکافی شود.

۲,۱۴,۲ راه‌حل سازگار

سازنده‌ی بدون آرگومان *SecureRandom* را، که از مقدار آغازین مشخص‌شده توسط سیستم برای تولید یک عدد تصادفی با طول ۱۲۸ بایت استفاده می‌کند، برگزینید.

```
byte[] randomBytes = new byte[128];
SecureRandom random = new SecureRandom();
random.nextBytes(randomBytes);
```

مشخص کردن دقیق تولیدکننده و فراهم‌کننده‌ی عدد تصادفی، کاری مناسب برای قابلیت حمل بهتر به‌شمار می‌رود.

۳,۱۴,۲ کاربرد

امن‌سازی اعداد تصادفی به میزان ناکافی، مهاجمان را قادر می‌سازد تا اطلاعات خاصی را در مورد مفهومی که در آن استفاده می‌شوند، به‌دست آورند. اعداد تصادفی ناامن، در مفهومی که نیازمند امنیت نیست، مفید هستند.

۱۵,۲ به روش‌هایی که نمی‌توانند توسط کد غیرقابل اعتماد بازنویسی شوند، تکیه نکنید

کد غیر قابل اعتماد می‌تواند از APIهای فراهم‌شده توسط کد مورداعتماد، برای بازنویسی متدهایی مانند *Object.equals()*، *Object.hashCode()*، و *Thread.run()*، سوءاستفاده کند. این متدها، اهداف با ارزشی هستند، زیرا معمولاً در پشت صحنه استفاده می‌شوند و ممکن است با مولفه‌ها، به‌صورتی که به‌سادگی قابل مشاهده نیست، ارتباط برقرار نمایند.

یک مهاجم می‌تواند با فراهم کردن پیاده‌سازی‌های لغوشده[۸۰]، از یک کد غیرقابل اعتماد برای جمع کردن اطلاعات حساس، اجرای کد دلخواه، یا انجام یک حمله‌ی انکار خدمت، استفاده کند.

۱,۱۵,۲ نمونه کد ناسازگار (*hashCode*)

این مثال، یک کلاس *LicenseManager* را نشان می‌دهد که یک *licenseMap* را نگه می‌دارد. این نقشه، یک *LicenseType* و یک جفت مقدار مجوز[۸۱] را ذخیره می‌کند.

سازنده‌ی *LicenseManager*، *licenseMap* را توسط یک کلید مجوز نمونه (دمو)[۸۲]، که باید سری باقی بماند، مقداردهی اولیه می‌کند. کلید مجوز، به‌منظور گویایی هرچه بیشتر، حک[۸۳] شده‌است؛ باید به‌صورت ایده‌آل، از یک فایل پیکربندی خارجی، که یک نسخه‌ی رمزنگاری‌شده از کلید را ذخیره می‌کند، خوانده شود. کلاس *LicenseType*، پیاده‌سازی‌های ملغی را از *equals()* و *hashCode()* فراهم می‌کند.

---

[۸۰] Overridden implementations

[۸۱] License

[۸۲] Demo

[۸۳] Hardcode

```
public class LicenseManager
{
	Map<LicenseType, String> licenseMap =new HashMap<LicenseType, String>();
	public LicenseManager()
	{
		LicenseType type = new LicenseType();
		type.setType("demo-license-key");
		licenseMap.put(type, "ABC-DEF-PQR-XYZ");
	}
	public Object getLicenseKey(LicenseType licenseType)
	{
		return licenseMap.get(licenseType);
	}
	public void setLicenseKey(LicenseType licenseType, String licenseKey)
	{
		licenseMap.put(licenseType, licenseKey);
	}
}
class LicenseType
{
	private String type;
	public String getType()
	{
		return type;
	}
	public void setType(String type)
	{
		this.type = type;
	}
	@Override
	public int hashCode()
	{
		int res = 17;
		res = res * 31 + type == null ? 0 : type.hashCode();
		return res;
	}
	@Override
	public boolean equals(Object arg)
	{
		if (arg == null || !(arg instanceof LicenseType)) {
			return false;
		}
		if (type.equals(((LicenseType) arg).getType())) {
			return true;
		}
		return false;
	}
}
```

این پیاده‌سازی، در برابر مهاجمی که کلاس *LicenseType* را توسعه می‌دهد و متدهای *equals()* و *hashCode()* را بازنویسی می‌نماید،آسیب‌پذیر است:

```
public class CraftedLicenseType extends LicenseType
{
	private static int guessedHashCode = 0;
	@Override
	public int hashCode()
	{
		// Returns a new hashCode to test every time get() is called
		guessedHashCode++;
		return guessedHashCode;
	}
	@Override
	public boolean equals(Object arg)
	{
		// Always returns true
		return true;
	}
}
```

در ادامه، برنامه‌ی مشتری مخرب آورده شده‌است.

```
public class DemoClient
{
	public static void main(String[] args)
	{
		LicenseManager licenseManager = new LicenseManager();
		for (int i = 0; i <= Integer.MAX_VALUE; i++)
		{
			Object guessed =
				licenseManager.getLicenseKey(new
				CraftedLicenseType());
			if (guessed != null)
			{
				// prints ABC-DEF-PQR-XYZ
				System.out.println(guessed);
			}
		}
	}
}
```

برنامه‌ی مشتری، از طریق دنباله‌ای از تمام کدهای درهم‌سازی ممکن و با استفاده از *CraftedLicenseType*، تا زمانی که به طور موفقیت‌آمیز با کد درهم‌سازی از شی کلید مجوز دموی ذخیره‌شده در کلاس *LicenseManager* مطابقت یابد، اجرا می‌شود. در نتیجه، مهاجم می‌تواند داده‌های حساس موجود در *licenseMap* را، تنها در چند دقیقه کشف نماید. حمله با کشف حداقل یک تداخل[۸۴] درهم‌سازی مرتبط با کلید نقشه، عمل می‌کند.

۲.۱۵.۲ راه‌حل سازگار (*IdentityHashMap*)

این راه‌حل، به جای *HashMap*، از *IdentityHashMap* برای ذخیره‌سازی اطلاعات مجوز استفاده می‌کند:

```
public class LicenseManager
{
      Map<LicenseType, String> licenseMap =
      new IdentityHashMap<LicenseType, String>();
      // ...
}
```

بنابر *کلاس IdentityHashMap* مستند Java API:

این کلاس، هنگام مقایسه‌ی دو کلید (و مقدار)، واسط *Map* حاوی یک جدول درهم‌سازی را با استفاده از یکسانی ارجاع[۸۵] به جای یکسانی شی، پیاده‌سازی می‌کند. به بیان دیگر، در یک *IdentityHashMap*، دو کلید *k1* و *k2* مساوی درنظر گرفته می‌شوند، اگر و تنها اگر (*k1==k2*) این در حالی است که در پیاده‌سازی‌های عادی Map (مانند *HashMap*)، دو کلید *k1* و *k2* مساوی درنظر گرفته می‌شوند، اگر و تنها اگر:

k1==null ? k2==null : k1.equals(k2)

در نتیجه، متدهای بازنویسی‌شده نمی‌توانند جزئیات کلاس داخلی را افشا نمایند. برنامه‌ی مشتری می‌تواند به افزودن کلیدهای مجوز ادامه دهد. حتی می‌تواند جفت‌های کلید-مقدار اضافه‌شده را، آن‌گونه که در کد مشتری زیر نشان داده شده‌است، بازیابی نماید.

---

[۸۴] Collision

[۸۵] Reference-equality

```
public class DemoClient
{
	public static void main(String[] args)
	{
		LicenseManager licenseManager = new LicenseManager();
		LicenseType type = new LicenseType();
		type.setType("custom-license-key");
		licenseManager.setLicenseKey(type, "CUS-TOM-LIC-KEY");
		Object licenseKeyValue = licenseManager.getLicenseKey(type);
		// Prints CUS-TOM-LIC-KEY
		System.out.println(licenseKeyValue);
	}
}
```

۳،۱۵،۲راه‌حل سازگار (*final Class*)

این راه‌حل، کلاس *LicenseType* را به‌عنوان *final* اعلان می‌کند تا متدهای آن نتوانند بازنویسی شوند:

```
final class LicenseType
{
	// ...
}
```

۴،۱۵،۲نمونه‌کد ناسازگار

این مثال، از یک کلاس *Widget* و یک کلاس *LayoutManager*، که حاوی مجموعه‌ای از ویجت‌ها است، تشکیل شده‌است:

```
public class Widget
{
	private int noOfComponents;
	public Widget(int noOfComponents)
	{
		this.noOfComponents = noOfComponents;
	}
	public int getNoOfComponents()
	{
		return noOfComponents;
	}
	public final void setNoOfComponents(int noOfComponents)
	{
		this.noOfComponents = noOfComponents;
	}
```

```
	public boolean equals(Object o)
	{
		if (o == null || !(o instanceof Widget))
		{
			return false;
		}
		Widget widget = (Widget) o;
		return this.noOfComponents == widget.getNoOfComponents();
	}
	@Override
	public int hashCode()
	{
		int res = 31;
		res = res * 17 + noOfComponents;
		return res;
	}
}
public class LayoutManager
{
	private Set<Widget> layouts = new HashSet<Widget>();
	public void addWidget(Widget widget)
	{
		if (!layouts.contains(widget))
		{
			layouts.add(widget);
		}
	}
	public int getLayoutSize()
	{
		return layouts.size();
	}
}
```

یک مهاجم می‌تواند کلاس *Widget* را توسعه دهد و متد *hashCode()* را بازنویسی کند:

```
public class Navigator extends Widget {
	public Navigator(int noOfComponents) {
		super(noOfComponents);
	}
	@Override
	public int hashCode() {
		int res = 31;
		res = res * 17;
		return res;
	}
}
```

کد مشتری به‌صورت زیر است:

```
Widget nav = new Navigator(1);
Widget widget = new Widget(1);
LayoutManager manager = new LayoutManager();
manager.addWidget(nav);
manager.addWidget(widget);
System.out.println(manager.getLayoutSize()); // Prints 2
```

انتظار می‌رود طرح‌بندی‌های[۸۶] مجموعه، تنها حاوی یک فقره[۸۷] باشند، زیرا تعداد مولفه‌هایی که برای هدایت‌گر[۸۸] و ویجت اضافه می‌شود، ۱ است. با این‌وجود، *getLayoutSize()*، ۲ را برمی‌گرداند.

دلیل این اختلاف این است که متد *hashCode()* کلاس *Widget*، تنها یک‌بار و هنگامی‌که ویجت به مجموعه اضافه می‌شود، مورد استفاده قرار می‌گیرد. هنگامی‌که هدایت‌گر اضافه می‌گردد، *hashCode()* فراهم‌شده توسط کلاس *Navigator*، استفاده می‌شود. متعاقباً، مجموعه، حاوی دو نمونه شی متفاوت خواهدبود.

۵,۱۵,۲راه‌حل سازگار (*final Class*)

این راه‌حل، کلاس *Widget* را *final* اعلان می‌کند تا نتوان روش‌ها را بازنویسی نمود:

```
public final class Widget
{
	// ...
}
```

۶,۱۵,۲نمونه‌کد ناسازگار (*run()*)

در این مثال، کلاس *Worker* و زیرکلاس آن، یعنی *SubWorker*، هریک شامل یک متد *startThread()* هستند که هدف آن، شروع یک نخ است.

---

[۸۶] Layouts

[۸۷] Item

[۸۸] Navigator

```
public class Worker implements Runnable
{
	Worker() { }
	public void startThread(String name)
	{
		new Thread(this, name).start();
	}
	@Override
	public void run()
	{
		System.out.println("Parent");
	}
}
public class SubWorker extends Worker
{
	@Override
	public void startThread(String name)
	{
		super.startThread(name);
		new Thread(this, name).start();
	}
	@Override
	public void run()
	{
		System.out.println("Child");
	}
}
```

اگر یک مشتری کد زیر را اجرا کند، ممکن است انتظار داشته باشد *Paernt* و *Child* چاپ شوند. اگرچه، *Child* دو بار چاپ می‌شود، زیرا متد *run()* بازنویسی‌شده، دو بار (هر دو بار، هنگام شروع یک نخ جدید) فراخوانی می‌گردد.

```
Worker w = new SubWorker();
w.startThread("thread");
```

۷.۱۵.۲راه‌حل سازگار

این راه‌حل، کلاس *SubWorker* را اصلاح، و فراخوانی *super.startThread()* را حذف می‌نماید:

```
public class SubWorker extends Worker
{
	@Override
	public void startThread(String name)
	{
		new Thread(this, name).start();
	}
	// ...
}
```

کد مشتری نیز اصلاح می‌شود تا نخ‌های والد و فرزند را به‌صورت جداگانه آغاز کند. این برنامه، خروجی مورد انتظار را تولید می‌نماید:

```
Worker w1 = new Worker();
w1.startThread("parent-thread");
Worker w2 = new SubWorker();
w2.startThread("child-thread");
```

۱۶,۲ از اعطای امتیازات اضافی اجتناب کنید

سیاست امنیت جاوا، این امکان را برای کد فراهم می‌آورد تا به منابع سیستمی خاصی دسترسی داشته‌باشد. یک *منبع کد* (یک شی از نوع *CodeSource*)، که به آن اجازه اعطا می‌شود، از مکان کد (URL) و یک ارجاعی به گواهی(هایی)[۸۹] حاوی کلید(های) عمومی مرتبط با کلید(های) خصوصی مورد استفاده برای امضای دیجیتال کد، تشکیل شده‌است. ارجاع به گواهی(ها)، تنها در صورتی مرتبط خواهدبود که کد، به‌صورت دیجیتالی امضا شده‌باشد. یک *دامنه‌ی حفاظتی*، شامل یک *CodeSource* و اجازه‌ی اعطاشده به کد از *CodeSource*، مطابق با سیاست امنیت تعیین‌شده‌ی در حال اجرای کنونی، است. در نتیجه، کلاس‌هایی که با کلید مشابهی امضا شده‌اند و از یک URL آمده‌اند، در یک دامنه‌ی حفاظتی مشابه قرار می‌گیرند. یک کلاس، فقط و فقط به یک دامنه‌ی حفاظتی تعلق دارد. کلاس‌هایی که اجازه‌های یکسانی دارند، اما از منابع کد متفاوتی هستند و به دامنه‌های متفاوتی تعلق دارند.

هر کلاس جاوا در دامنه‌ی مناسب خود، که توسط منبع کد آن تعیین شد، اجرا می‌شود. برای هر کدی که تحت مدیریت امنیتی اجرا می‌شود تا یک عمل امن، مانند خواندن یا نوشتن یک فایل، را انجام دهد، باید به کد مذکور اجازه‌ی انجام آن عمل خاص اعطا شود. کد ممتاز می‌تواند با استفاده از متد *AccessController.doPrivileged()*، به نیابت از یک فراخواننده‌ی غیرممتاز، به منابع ممتاز دسترسی پیدا کند. به‌عنوان نمونه، این امر زمانی ضروری است که یک ابزار[۹۰] سیستم، به باز کردن یک فایل فونت از طرف کاربر نیاز دارد تا یک مستند را نمایش دهد، اما برنامه اجازه‌ی انجام چنین کاری را ندارد. برای انجام این کار، ابزار سیستم از امتیازات کامل خود برای به‌دست آوردن فونت‌ها استفاده می‌کند و امتیازات فراخواننده را نادیده می‌گیرد. کد ممتاز، با تمام امتیازات دامنه‌ی حفاظتی مرتبط با منبع کد، اجرا می‌شود. این امتیازات، اغلب بیش از امتیازات لازم برای انجام عملیات ممتاز هستند. به‌طور ایده‌آل، تنها باید کمترین مجموعه از امتیازت لازم برای انجام عملیات، به کد اعطا شود.

---

[۸۹] Certificate

[۹۰] Utility

۱.۱۶.۲ نمونه کد ناسازگار

این مثال، یک متد کتابخانه را نشان می‌دهد که به فراخواننده‌ها اجازه می‌دهد تا عملیات ممتاز (خواندن یک فایل) را با استفاده از متد پنهان‌سازی *performActionOnFile()* انجام دهند:

```
private FileInputStream openFile()
{
	final FileInputStream f[] = { null };
	AccessController.doPrivileged(new PrivilegedAction()
				{
					public Object run()
					{
						try
						{
							f[0] = new
							FileInputStream("file");
						}
						catch(FileNotFoundException fnf)
						{
							// Forward to handler
						}
						return null;
					}
				});
	return f[0];
}
// Wrapper method
public void performActionOnFile()
{
	try (FileInputStream f = openFile())
	{
		// Perform operation
	}
	catch (Throwable t)
	{
		// Handle exception
	}
}
```

در این مثال، کد قابل اعتماد، حتی اگر دستیابی خواندنی به فایل تنها اجازه‌ی مورد نیاز توسط بلاک doPrivileged() باشد، امتیازاتی بیش از امتیازات مورد نیاز برای خواندن یک فایل را اعطا می‌کند. در نتیجه، این کد، اصل حداقل امتیاز را با فراهم کردن امتیازات غیرضروری و زائد برای بلاک کد، نقض می‌کند.

۲,۱۶,۲ راه‌حل سازگار

فرم دو آرگومانی *doPrivileged()*، یک شی *AccessControlContext* را از فراخواننده می‌پذیرد و امتیازات کد مشمول را به‌عنوان آرگومان دوم، به تقاطع امتیازات دامنه‌ی حفاظتی و امتیازات مفهوم ارسال‌شده، محدود می‌کند. در نتیجه، فراخواننده‌ای که تمایل دارد تنها اجازه‌ی خواندن فایل را اعطا کند، می‌تواند مفهومی را فراهم کند که تنها، اجازه‌های مربوط به خواندن فایل را در اختیار دارد.

یک *AccessControlContext*، که اجازه‌های مناسب برای خواندن فایل را در اختیار دارد، می‌تواند به‌عنوان یک کلاس درونی ساخته شود:

```
private FileInputStream openFile(AccessControlContext context)
{
	if (context == null)
	{
		throw new SecurityException("Missing AccessControlContext");
	}
	final FileInputStream f[] = { null };
	AccessController.doPrivileged(new PrivilegedAction()
				{
					public Object run()
					{
						try
						{
							f[0] = new
							FileInputStream("file");
						}
						catch (FileNotFoundException fnf)
						{
							// Forward to handler
						}
						return null;
					}
				},
				// Restrict the privileges by passing the
				// context argument
				context);
	return f[0];
}
```

```
private static class FileAccessControlContext
{
	public static final AccessControlContext INSTANCE;
	static
	{
		Permission perm = new java.io.FilePermission("file", "read");
		PermissionCollection perms = perm.newPermissionCollection();
		perms.add(perm);
		INSTANCE = new AccessControlContext(new ProtectionDomain[]
							{
								new ProtectionDomain(null,
								perms)
							});
	}
}
// Wrapper method
public void performActionOnFile()
{
	try (final FileInputStream f = // Grant only open-for-reading privileges
		openFile(FileAccessControlContext.INSTANCE))
		{
			// Perform action
		}
		catch (Throwable t)
		{
			// Handle exception
		}
}
```

فراخواننده‌هایی که اجازه‌ی ساخت یک *AccessControlContext* را ندارند، می‌توانند با استفاده از *AccessController.getContext()*، برای ساخت نمونه، درخواست دهند.

۳.۱۶.۲ کاربرد

شکست در دنبال کردن اصل حداقل امتیاز، می‌تواند منجر به کدی غیرقابل اعتماد و غیرممتازی شود که عملیات ناخواسته‌ی ممتاز انجام می‌دهد. با این‌وجود، محدود کردن با دقت امتیازات، پیچیدگی را افزایش می‌دهد. این پیچیدگی افزایش‌یافته و کاهش قابلیت نگهداری مرتبط با آن، باید در برابر هر بهبود امنیتی مصالحه شود.

۱۷,۲ کدهای ممتاز را کمینه کنید

برنامه‌ها باید با اصل حداقل امتیاز، نه فقط با فراهم کردن بلاک‌های ممتاز با کمترین اجازه‌های مورد نیاز برای درست عمل کردن، بلکه با حصول اطمینان از این‌که کد ممتاز، تنها حاوی آن عملیاتی باشد که نیاز به امتیازت افزایش‌یافته دارد، مطابق باشند. کد زائد و غیرضروری که درون یک بلاک ممتاز قرار دارد، باید با امتیازات آن بلاک کار کند، که این امر، منجر به افزایش سطح حمله می‌شود.

1.17.2 نمونه‌کد ناسازگار

این مثال، حاوی یک *changePassword()* است که تلاش می‌کند یک فایل رمز عبور را درون بلاک *doPrivileged* باز نماید و با استفاده از آن فایل، عملیاتی را انجام دهد. بلاک *doPrivileged*، حاوی یک فراخوانی *System.loadLibrary()* اضافی نیز است که کتابخانه‌ی احراز اصالت را بار می‌کند.

```
public void changePassword(String currentPassword,String newPassword) {
	final FileInputStream f[] = { null };
	AccessController.doPrivileged(
		new PrivilegedAction()
		{
			public Object run()
			{
				try
				{
					String passwordFile =
					System.getProperty("user.dir") +
					File.separator + "PasswordFileName";
					f[0] = new FileInputStream(passwordFile);
					// Check whether oldPassword matches the one in
					// the file. If not, throw an exception
					System.loadLibrary("authentication");
				}
				catch (FileNotFoundException cnf)
				{
					// Forward to handler
				}
				return null;
			}
		}); // End of doPrivileged()
}
```

این مثال، اصل حداقل امتیاز را نقض می‌کند، زیرا یک فراخواننده‌ی غیرممتاز نیز می‌تواند کتابخانه‌ی احراز اصالت را بار کند. یک فراخواننده‌ی غیرممتاز نمی‌تواند مستقیما متد *System.loadLibrary()* را فراخوانی نماید، زیرا این کار می‌تواند متدهای اصیل[۹۱] را در معرض کد غیرممتاز قرار دهد. علاوه بر این، متد *System.loadLibrary()* فقط امتیازات فراخواننده‌ی بی‌درنگ خود را بررسی می‌کند. بنابراین، تنها باید با احتیاط زیاد از آن استفاده شود.

۲.۱۷.۲ راه‌حل سازگار

این راه‌حل، فراخوانی *System.loadLibrary()* را به خارج از بلاک *doPrivileged()* منتقل می‌کند. انجام این‌کار، به کد غیر ممتاز اجازه می‌دهد تا بررسی‌های اولیه‌ی بازنشانی رمز عبور[۹۲] را با استفاده از فایل انجام دهد، اما از بار کردن کتابخانه‌ی احراز اصالت توسط آن جلوگیری به‌عمل می‌آورد.

```
public void changePassword(String currentPassword,String newPassword)
{
	final FileInputStream f[] = { null };
	AccessController.doPrivileged(
	new PrivilegedAction()
	{
		public Object run()
		{
			try
			{
				String passwordFile = System.getProperty("user.dir")
				+ File.separator + "PasswordFileName";
				f[0] = new FileInputStream(passwordFile);
				// Check whether oldPassword matches the one in the
				// file. If not, throw an exception
			}
			catch (FileNotFoundException cnf)
			{
				// Forward to handler
			}
			return null;
		}
	}); // End of doPrivileged()
	System.loadLibrary("authentication");
}
```

[۹۱] Native

[۹۲] Password-reset

فراخوانی *loadLibrary()* می‌تواند پیش از انجام بررسی‌های بازنشانی رمز عبور نیز رخ دهد. در این مثال، به دلیل کارایی، معوق[۹۳] شده‌است.

۳.۱۷.۲ کاربرد

کمینه‌سازی کد ممتاز، سطح حمله‌ی یک برنامه‌ی کاربردی را کاهش می‌دهد و وظیفه‌ی حسابرسی کد ممتاز را ساده می‌سازد.

[۹۳] Deferred

۱۸,۲ روش‌هایی را که از بررسی‌های امنیتی کاهش‌یافته استفاده می‌کنند، در معرض کدهای غیرقابل اعتماد قرار ندهید

اغلب روش‌ها، فاقد بررسی‌های مدیریت امنیت هستند، زیرا دسترسی به بخش‌های حساس سیستم، مانند فایل سیستم، را فراهم نمی‌آورند. بیشتر روش‌هایی که بررسی‌های مدیریت امنیت را مهیا می‌کنند، پیش از ادامه دادن وارسی می‌نمایند که هر کلاس و متد در پُشته‌ی[۹۴] فراخوانی، مجاز باشد. این مدل امنیتی، به برنامه‌ی محدود، مانند اَپلت‌های جاوا، اجازه می‌دهد تا به هسته‌ی کتابخانه‌ی جاوا، دسترسی کامل داشته‌باشند. همچنین، جلوی عملکرد یک متد حساس را به جای یک متد مخرب، که پشت متدهای قابل اعتماد در پُشته‌ی فراخوانی پنهان می‌شود، می‌گیرد. با این وجود، متدهای مشخصی از بررسی امنیتی کاهش‌یافته استفاده می‌کنند. این متدها، تنها بررسی می‌نمایند که متد فراخواننده (و نه تمام متدهای موجود در پُشته‌ی فراخوانی)، مجاز باشد. هر کدی که این روش‌ها را فراخوانی کند، باید تضمین نماید که نمی‌توان آنها را از طرف یک کد غیرقابل اعتماد، فراخوانی نمود. این متد در جدول ۲-۲ فهرست شده‌اند.

جدول ۲-۲: متدهایی که تنها، متد فراخواننده را بررسی می‌کنند.

java.lang.Class.newInstance
java.lang.reflect.Constructor.newInstance
java.lang.reflect.Field.get*
java.lang.reflect.Field.set*
java.lang.reflect.Method.invoke
java.util.concurrent.atomic.AtomicIntegerFieldUpdater.newUpdater
java.util.concurrent.atomic.AtomicLongFieldUpdater.newUpdater
java.util.concurrent.atomic.AtomicReferenceFieldUpdater.newUpdater

چون متدهای *java.lang.reflect.Field.setAccessible()* و *getAccessible()* برای ساخت JVM، جهت بازنویسی بررسی‌های دسترسی استفاده می‌شوند، بررسی‌های مدیریت امنیت استاندارد (و محدود کننده‌تری) انجام می‌دهند. در نتیجه، فاقد آسیب‌پذیری‌های مطرح‌شده توسط این راهنما هستند. با این‌وجود، این متدها نیز باید با دقت فراوان استفاده شوند. متدهای انعکاس فیلد *set** و *get** باقیمانده، تنها، بررسی‌های دسترسی زبان را انجام می‌دهند و از این‌روی، آسیب‌پذیر هستند.

---

[۹۴] Stack

1.18.2 بارگذاران کلاس

بارگذاران کلاس، به یک برنامه‌ی کاربردی جاوا اجازه می‌دهند تا به‌صورت پویا در زمان اجرا، توسط کلاس‌های اضافی، توسعه یابد. JVM برای هر کلاسی که بار می‌شود، بارگذار کلاس را، که برای بارگذاری این کلاس استفاده شده‌بود، دنبال می‌نماید. هنگامی‌که یک کلاس بارگذاری‌شده برای اولین بار به کلاس دیگری ارجاع می‌دهد، ماشین مجازی درخواست می‌کند که کلاس مورد ارجاع، توسط همان بارگذار کلاسی که برای کلاس ارجاع‌کننده استفاده شده‌بود، بار شود. معماری بارگذار کلاس جاوا، تعاملات میان کد بارشده را با تجویز استفاده از بارگذاران مختلف کلاس، از منابع مختلف کنترل می‌کند. این جداسازی بارگذاران کلاس، برای جداسازی کد، امری بنیادی است: از دست‌یابی کد مخرب و خراب‌کاری آن در کد قابل اعتماد، ممانعت به‌عمل می‌آورد.

متدهای گوناگونی که مسئول کلاس‌های بارگذار هستند، کار خود را به بارگذار کلاس متدی که آنها را فراخوانی کرده‌است، محول می‌کنند. بررسی‌های امنیتی مرتبط با کلاس‌های بارگذار، توسط بارگذاران کلاس انجام می‌پذیرد. در نتیجه، هر روشی که یکی از این متدهای بارگذاری کلاس را فراخوانی نماید، باید تضمین کند که این متدها نمی‌توانند از جانب کد غیرقابل اعتماد عمل نمایند. این متدها در جدول ۲-۳ فهرست شده‌اند.

جدول ۲-۳: متدهایی که از بارگذاران کلاس متدهای فراخوانده استفاده می‌کنند.

java.lang.Class.forName
java.lang.Package.getPackage
java.lang.Package.getPackages
java.lang.Runtime.load
java.lang.Runtime.loadLibrary
java.lang.System.load
java.lang.System.loadLibrary
java.sql.DriverManager.getConnection
java.sql.DriverManager.getDriver
java.sql.DriverManager.getDrivers
java.sql.DriverManager.deregisterDriver
java.utl.ResourceBundle.getBundle

به استثنای متدهای *loadLibrary()* و *load()*، متدهای درون جدول، هیچ بررسی مدیریت امنیتی انجام نمی‌دهند؛ آنها بررسی‌های امنیت را به بارگذاران کلاس مناسب محول می‌کنند.

در عمل، بار کننده‌های کلاس این کدهای قابل اعتماد، به متدهای مذکور اجازه می‌دهند تا فراخوانی شوند. این درحالی است که ممکن است بارگذاران کلاس کدهای غیرقابل، فاقد این امتیازات باشند. با این‌وجود، هنگامی‌که بارگذار کلاس کد غیرقابل اعتماد، به بارگذار کلاس کد قابل اعتماد محول می‌شود، کد غیرقابل اعتماد به کد قابل اعتماد، دیدی به‌دست می‌آورد. در غیاب چنین رابطه‌ای برای محول‌سازی، بارگذاران کلاس باید از جداسازی فضای‌نام[۹۵] اطمینان حاصل کنند. در نتیجه، کد غیرقابل اعتماد، قادر به مشاهده‌ی اعضا یا فراخوانی متدهای متعلق به کد قابل اعتماد، نخواهد بود.

مدل محول‌سازی بارگذار کلاس، برای بسیاری از چارچوب‌ها و پیاده‌سازی‌های جاوا اساسی است. از افشای متدهای فهرست‌شده در جداول ۲-۲ و ۲-۳ به کدهای غیرقابل اعتماد، اجتناب کنید. به‌عنوان نمونه، یک سناریوی حمله را درنظر بگیرید که در آن، کد غیرقابل اعتماد تلاش می‌کند تا یک کلاس ممتاز را بار نماید. اگر بارگذار کلاس آن، به خودی خود فاقد اجازه‌ی بارگذاری کلاس ممتاز مورد تقاضا باشد، اما بارگذار کلاس، اجازه‌ی محول‌سازی بارگذاری کلاس را به بارگذار یک کلاس قابل اعتماد بدهد، ممکن است افزایش امتیاز رخ دهد.

علاوه بر این، اگر کد قابل اعتماد، ورودی‌های آلوده[۹۶] را بپذیرد، بارگذار کلاس کد قابل اعتماد می‌تواند متقاعد شود تا کلاس‌های ممتازِ مخرب را به نیابت از کد غیرقابل اعتماد، بارگذاری نماید.

کلاس‌هایی که دارای همان بارگذار کلاس تعریف‌کننده هستند، در همان فضای‌نام وجود خواهند داشت، اما می‌توانند بسته به سیاست امنیت، امتیازات مختلفی داشته‌باشند. آسیب‌پذیری‌های امنیتی می‌توانند زمانی به وجود آیند که کد ممتاز با کد غیرممتازی (یا کد با امتیاز کمتر) هم‌زیستی داشته باشد که توسط همان بارگذار کلاس، بارگذاری شده‌است. در این وضعیت، کد با امتیاز کمتر می‌تواند طبق قابلیت دسترسی اعلان شده توسط کد ممتاز، آزادانه به اعضای کد ممتاز، دسترسی داشته‌باشد. هنگامی‌که کد ممتاز از هر یک از APIهای جدول استفاده می‌کند، بررسی‌های مدیریت امنیت (به استثنای *loadLibrary()* و *load()*) را دور می‌زند (SEC03-J. کلاس‌های قابل اعتماد را پس از اجازه دادن به کد غیرقابل اعتماد برای

---

[۹۵] Namespace

[۹۶] Tainted

بارگذاری کلاس‌های دلخواه، بارگذاری نکنید). همچنین، بسیاری از مثال‌ها، "SEC00-J. به بلاک‌های ممتاز اجازه ندهید اطلاعات حساس را در یک مرز اعتماد نشت دهند"، را نقض می‌کنند.

۲.۱۸.۲ نمونه کد ناسازگار

در این مثال، یک فراخوانی *System.loadLibrary()* درون بلاک *doPrivileged* تعبیه شده‌است.

```
public void load(String libName)
{
	AccessController.doPrivileged(
		new PrivilegedAction()
		{
			public Object run()
			{
				System.loadLibrary(libName);
				return null;
			}
		});
}
```

این کد، ناامن است، زیرا می‌تواند برای بارگذاری یک کتابخانه، به نیابت از کد غیرقابل اعتماد استفاده شود. در اصل، ممکن است بارگذار کلاس کد غیرقابل اعتماد، قادر به استفاده از این کد برای بارگذاری یک کتابخانه، حتی با عدم وجود اجازه‌های کافی برای انجام مستقیم چنین کاری، باشد. پس از بار کردن کتابخانه، کد غیرقابل اعتماد می‌تواند متدهای اصیل را، در صورتی‌که قابل وصول باشند، از کتابخانه فراخوانی نماید، زیرا بلاک *doPrivileged*، از انجام هر بررسی مدیریت امنیتی روی فراخواننده‌ها تا پُشته‌ی اجرایی، جلوگیری می‌کند.

کد کتابخانه‌ی غیراصیل می‌تواند مستعد اشکالات امنیتی مرتبط باشد. فرض کنید کتابخانه‌ای وجود دارد که حاوی یک آسیب‌پذیری است و احتمالاً، به‌دلیل آن‌که در یک متد استفاده‌نشده واقع شده‌است، مستقیما در معرض آن قرار ندارد. ممکن است بارگذاری این کتابخانه، مستقیما نشان‌دهنده‌ی یک آسیب‌پذیری نباشد. با این‌حال، یک مهاجم می‌تواند یک کتابخانه‌ی اضافی را بار کند تا از آسیب‌پذیری کتابخانه‌ی اول سوءاستفاده نماید. علاوه بر این، معمولاً کتابخانه‌های غیراصیل، از بلاک‌های *doPrivileged* استفاده می‌کنند، که آنها را به اهدافی جذاب تبدیل می‌کند.

۳,۱۸,۲راه‌حل سازگار

این راه‌حل، نام کتابخانه را حک می‌کند تا جلوی احتمال آلودگی مقادیر را بگیرد. همچنین، قابلیت دسترسی به *load()* را از عمومی به خصوصی کاهش می‌دهد. در نتیجه، فراخواننده‌های غیرقابل اعتماد، از بارگذاری کتابخانه‌ی *awt* منع می‌شوند.

```
private void load()
{
	AccessController.doPrivileged(
		new PrivilegedAction()
		{
			public Object run()
			{
				System.loadLibrary("awt");
				return null;
			}
		});
}
```

۴,۱۸,۲نمونه‌کد ناسازگار

این مثال، نمونه‌ای از *java.sql.Connection* را، از کد قابل اعتماد به کد غیرقابل اعتماد بازمی‌گرداند.

```
public Connection getConnection(String url, String username,String password)
{
	// ...
	return DriverManager.getConnection(url, username, password);
}
```

کد غیرقابل اعتماد، فاقد اجازه‌های لازم برای ساخت یک ارتباط SQL است که بتواند این محدودیت‌ها را مستقیما با استفاده نمونه‌ی به‌دست‌آمده، دور بزند. متد *getConnection()* غیرایمن است، زیرا از آرگومان URL برای نشان دادن کلاسی که باید بارگذاری شود، استفاده می‌کند. این کلاس، به‌عنوان درایور پایگاه داده عمل می‌کند.

۵,۱۸,۲راه‌حل سازگار

این راه‌حل، به کاربران مخرب اجازه نمی‌دهد تا URL خود را برای ارتباط با پایگاه داده به کار برند. از این‌روی، قابلیت آنها را برای بارگذاری درایورهای غیرقابل اعتماد، محدود می‌سازد.

```
private String url = // Hardwired value
public Connection getConnection(String username,String password)
{
      // ...
      return DriverManager.getConnection(this.url, username, password);
}
```

۶,۱۸,۲نمونه‌ک ناسازگار (CERT Vulnerability 636312)

یادداشت آسیب‌پذیری CERT، با عنوان VU#636312، یک آسیب‌پذیری را در به‌روزرسانی ششم Java 1.7.0، توصیف می‌کند که در ماه آگوست سال ۲۰۱۲، به‌طور گسترده مورد سوءاستفاده قرار گرفت. در حقیقت، این سوءاستفاده، از دو آسیب‌پذیری استفاده نمود (SEC05-J. از انعکاس، برای افزایش قابلیت دسترسی کلاس‌ها، روش‌ها، یا فیلدها استفاده نکنید).

```
private String url = // Hardwired value
public Connection getConnection(String username, String password)
{
      // ...
      return DriverManager.getConnection(this.url, username, password);
}
```

این سوءاستفاده، یک اَپلت جاوا را اجرا می‌کند. بارگذار کلاس اَپلت، اطمینان می‌یابد که یک اَپلت نمی‌تواند مستقیما متدهای کلاس‌های ارائه‌شده در پکیج *com.sun.* را فراخوانی کند. یک بررسی مدیریت امنیت عادی، از این‌که عمل‌های خاصی بسته به امتیازات تمام متدهای فراخواننده در پُشته‌ی فراخوانی، مجاز هستند یا رد می‌شوند، اطمینان حاصل می‌کند. لازم به ذکر است که امتیازات، مرتبط با منبع کدی هستند که حاوی کلاس است.

اولین هدف سوءاستفاده از کد، دسترسی به کلاس *sun.awt.SunToolkit* بود. با این‌وجود، فراخوانی مستقیم *class.forName()* روی نام این کلاس، منجر به ایجاد یک *SecurityException* می‌شود. در نتیجه، کد سوءاستفاده، از متد زیر برای دسترسی به هر کلاسی و دور زدن مدیریت امنیت، استفاده می‌کند:

```
private Class GetClass(String paramString)
throws Throwable
{
	Object arrayOfObject[] = new Object[1];
	arrayOfObject[0] = paramString;
	Expression localExpression =
		new Expression(Class.class, "forName", arrayOfObject);
	localExpression.execute();
	return (Class)localExpression.getValue();
}
```

متد *java.beans.Expression.execute()*، کار خود را به متد زیر محول می‌کند:

```
private Object invokeInternal() throws Exception
{
	Object target = getTarget();
	String methodName = getMethodName();
	if (target == null || methodName == null)
	{
		throw new NullPointerException((target == null ? "target" :
			"methodName") + " should not be null");
	}
	Object[] arguments = getArguments();
	if (arguments == null)
	{
		arguments = emptyArray;
	}
	// Class.forName() won't load classes outside of core from a class inside
	// core, so it is handled as a special case.
	if (target == Class.class && methodName.equals("forName"))
	{
		return ClassFinder.resolveClass((String)arguments[0],this.loader);
	}
	// ...
```

متد *com.sun.beans.finder.ClassFinder.resolveClass()*، کار خود را به *findClass()* محول می‌نماید:

```
public static Class<?> findClass(String name)
throws ClassNotFoundException
{
	try
	{
		ClassLoader loader =
			Thread.currentThread().getContextClassLoader();
		if (loader == null)
		{
			loader = ClassLoader.getSystemClassLoader();
		}
		if (loader != null)
		{
			return Class.forName(name, false, loader);
		}
	}
	catch (ClassNotFoundException exception)
	{
		// Use current class loader instead
	}
	catch (SecurityException exception)
	{
		// Use current class loader instead
	}
	return Class.forName(name);
}
```

اگرچه، این متد در مفهوم اَپلت فراخوانی می‌شود، از *Class.forName()* برای به‌دست آوردن کلاس موردتقاضا استفاده می‌کند. *Class.forName()*، جست‌وجو را به بارگذاری کلاس متد فراخواننده، محول می‌سازد. در این حالت، کلاس فراخواننده (*com.sun.beans.finder.ClassFinder*) بخشی از هسته‌ی جاوا می‌شود، بنابراین، بارگذار کلاس قابل اعتماد، به جای بارگذار کلاس محدودتر اَپلت، استفاده می‌شود و بارگذاری کلاس قابل اعتماد، بدون آگاهی از این‌که به نیابت از یک کد مخرب عمل می‌کند، کلاس‌ها را بارگذاری می‌نماید.

۷٫۱۸٫۲راه‌حل سازگار (CVE-2012-4681)

اوراکل، این آسیب‌پذیری را در به‌روزرسانی هفتم Java 1.7.0، با الصاق[۹۷] متد *com.sun.beans.finder.ClassFinder.findClass()*، کاهش داد. متد *checkPackageAccess()*، تمام پُشته‌ی فراخوانی را بررسی می‌کند تا مطمئن شود *Class.forName()*، تنها در این نمونه، کلاس‌ها را فقط از طرف متدهای قابل اعتماد، واکشی می‌نماید.

```
public static Class<?> findClass(String name)
throws ClassNotFoundException
{
	checkPackageAccess(name);
	try
	{
		ClassLoader loader=Thread.currentThread().getContextClassLoader();
		if (loader == null)
		{
			// Can be null in IE (see 6204697)
			loader = ClassLoader.getSystemClassLoader();
		}
		if (loader != null)
		{
			return Class.forName(name, false, loader);
		}
	}
	catch (ClassNotFoundException exception)
	{
		// Use current class loader instead
	}
	catch (SecurityException exception)
	{
		// Use current class loader instead
	}
	return Class.forName(name);
}
```

۸٫۱۸٫۲نمونه‌کد ناسازگار (CVE-2013-0422)

به‌روزرسانی دهم از Java 1.7.0، به‌دلیل چند آسیب‌پذیری، به‌صورت گسترده در ماه ژوئن سال ۲۰۱۳ مورد سوءاستفاده قرار گرفت. یکی از این آسیب‌پذیری‌ها، در کلاس

[۹۷] Patching

com.sun.jmx.mbeanserver.MbeanInstantiator، قابلیت دستیابی به هر کلاسی را، صرف‌نظر از سیاست فعلی امنیت یا قوانین قابلیت دسترسی، به کد غیرقابل اعتماد اعطا می‌کرد. متد *MBeanInstantiator.findClass()* می‌توانست با هر رشته‌ای فراخوانی شود و تلاش می‌کرد شی *Class* را، که پس از رشته، نام‌گذاری شده‌بود، برگرداند. این متد، کار خود را به متد *MBeanInstantiator.loadClass()* محول می‌کند که کد آن در زیر آورده شده‌است:

```
/**
* Load a class with the specified loader, or with this object
* class loader if the specified loader is null.
**/
static Class<?> loadClass(String className, ClassLoader loader)
throws ReflectionException
{
       Class<?> theClass;
       if (className == null)
       {
              throw new RuntimeOperationsException(
                   new IllegalArgumentException("The class name cannot be null"),
                     "Exception occurred during object instantiation");
       }
       try
       {
              if (loader == null)
              {
                     loader = MBeanInstantiator.class.getClassLoader();
              }
              if (loader != null)
              {
                     theClass = Class.forName(className, false, loader);
              }
              else
              {
                     theClass = Class.forName(className);
              }
       }
       catch (ClassNotFoundException e)
       {
         throw new ReflectionException(e, "The MBean class could not be loaded");
       }
       return theClass;
}
```

این متد، وظیفه‌ی بارگذاری پویای کلاس مشخص‌شده را، به متد *Class.forName()* محول می‌کند، که آن هم، کار را به بارگذار کلاس روش فراخواننده‌ی خود می‌سپارد. از آنجاکه *MBeanInstantiator.loadClass()*، متد فراخواننده است، از بارگذار کلاس هسته، که هیچ بررسی امنیتی را تعبیه نمی‌نماید، استفاده می‌شود.

۹،۱۸،۲راه‌حل سازگار (CVE-2013-0422)

اوراکل، این آسیب‌پذیری را با افزودن یک بررسی دسترسی به متد *MBeanInstantiator.loadClass()*، در به‌روزرسانی یازدهم Java 1.7.0 کاهش داد. این بررسی دسترسی، تضمین می‌نماید فراخواننده، اجازه‌ی دسترسی به کلاسی را که مورد جست‌وجو است، داشته باشد:

```
// ...
if (className == null)
{
	throw new RuntimeOperationsException(
		new IllegalArgumentException("The class name cannot be null"),
			"Exception occurred during object instantiation");
}
ReflectUtil.checkPackageAccess(className);
try
{
	if (loader == null)
	// ...
```

۱۰،۱۸،۲ کاربرد

اجازه دادن به کدهای غیرقابل اعتماد برای فراخوانی روش‌هایی با بررسی‌های امنیتی کاهش‌یافته، می‌تواند منجر به افزایش امتیاز شود. به طور مشابه، اجازه دادن به کدهای غیرقابل اعتماد برای انجام عملیات با استفاده از بارگذار کلاس فراخواننده‌ی بی‌درنگ، می‌تواند این امکان را برای کد مذکور فراهم آورد تا با همان امتیازات فراخواننده‌ی بی‌درنگ، اجرا شود.

روش‌هایی که از نمونه‌ی بارگذار کلاس فراخواننده‌ی بی‌درنگ اجتناب می‌کنند، خارج از حوزه‌ی این مستند هستند. به‌عنوان نمونه، متد سه آرگومانی *java.lang.Class.forName()*، به یک آرگومان صریح، که نمونه‌ی بارگذار کلاس را برای استفاده مشخص می‌کند، نیاز دارد.

```
public static Class forName(String name, boolean initialize,
      ClassLoader loader) throws ClassNotFoundException
```

هنگامی‌که نمونه‌ها باید به کد غیرقابل اعتماد برگردانده شوند، از بارگذار کلاس فراخواننده‌ی بی‌درنگ، به‌عنوان سومین آرگومان استفاده نکنید.

۱۹,۲ مجوزهای امنیت سفارشی را برای امنیت خوب تعریف کنید

SecurityManager پیش‌فرض، بررسی می‌کند که آیا فراخواننده‌ی یک روش خاص، مجوزهای کافی را برای انجام یک عمل در اختیار دارد یا خیر. یک عمل در معماری امنیت جاوا، به‌عنوان سطحی از دسترسی، تعریف شده‌است و به مجوزهای خاصی پیش از انجام آن نیاز دارد. به‌عنوان نمونه، اعمال *java.io.FilePermission*، شامل خواندن، نوشتن، اجرا، و حذف هستند.

گاهی، محدودیت‌های قوی‌تر از نمونه‌هایی که توسط مدیر امنیت به‌صورت پیش‌فرض فراهم شده‌اند، ضروری است. شکست در فراهم نمودن مجوزهای سفارشی[۹۸]، هنگامی‌که هیچ مجوز پیش‌فرضی وجود ندارد، می‌تولند منجر به آسیب‌پذیری‌های بالا بردن امتیاز شود که فراخواننده‌های غیرقابل اعتماد را قادر می‌سازد تا عملیات محدودشده اجرا نمایند. این بخش، به مساله‌ی امتیازات اضافی می‌پردازد.

۱,۱۹,۲ نمونه کد ناسازگار

این مثال، حاوی یک بلاک ممتاز است که برای انجام دو عمل حساس استفاده می‌شود: بار کردن یک کتابخانه؛ و تنظیم پیش‌فرض اداره‌کننده‌ی استثنا.

```
class LoadLibrary
{
	private void loadLibrary()
	{
		AccessController.doPrivileged(
			new PrivilegedAction()
			{
				public Object run()
				{
					// Privileged code
					System.loadLibrary("myLib.so");
					// Perform some sensitive operation like
					// setting the default exception handler
					MyExceptionReporter.setExceptionReporter(repor
					ter);
					return null;
				}
			});
}
}
```

[۹۸] custom

هنگامی‌که مورد استفاده قرار می‌گیرد، مدیر امنیت پیش‌فرض، بارگذاری کتابخانه را ممنوع می‌کند، مگر این‌که، *RuntimePermission loadLibrary.myLib*، در فایل سیاست[۹۹] اعطا شود. با این‌وجود، مدیر امنیت به‌طور خودکار، از یک فراخواننده در انجام دومین عمل حساس، که تنظیم پیش‌فرض اداره‌کننده‌ی استثنا است، محافظت نمی‌کند، زیرا مجوز این عمل پیش‌فرض، موجود نیست. از این ضعف امنیتی می‌توان به طرق مختلف، از قبیل برنامه‌نویسی و نصب یک اداره‌کننده‌ی استثنا، که اطلاعاتی را که یک اداره‌کننده فیلتر می‌کند، آشکار می‌سازد، سوءاستفاده نمود.

۲.۱۹.۲ راه‌حل سازگار

این راه‌حل، یک مجوز سفارشی *ExceptionReporterPermission* را با هدف *exc.reporter* تعریف می‌کند تا از تنظیم پیش‌فرض اداره‌کننده‌ی استثنا توسط فراخواننده‌های غیرقانونی، ممانعت به‌عمل آورد. این‌کار را می‌توان با زیرکلاس‌سازی از *BasicPermission* انجام داد که مجوزهای *binary*-style (یا اجازه دادن یا ممنوع کردن) را اعطا می‌کند. سپس راه‌حل سازگار، از مدیر امنیت استفاده می‌کند تا بررسی نماید که آیا فراخواننده، دارای مجوز لازم برای تنظیم اداره‌کننده است یا خیر. اگر بررسی با شکست مواجه شود، این کد، یک *SecurityException* ایجاد می‌نماید. کلاس مجوز سفارشی *ExceptionReporterPermission* نیز با دو سازنده‌ی مورد نیاز تعریف می‌شود.

[۹۹] Policy file

```
class LoadLibrary
{
	private void loadLibrary()
	{
		AccessController.doPrivileged(
			new PrivilegedAction() {
				public Object run()
				{
					// Privileged code
					System.loadLibrary("myLib.so");
					// Perform some sensitive operation like
					// setting the default exception handler
					MyExceptionReporter.setExceptionReporter(repor
					ter);
					return null;
				}
			});
	}
}
final class MyExceptionReporter extends ExceptionReporter
{
	public void setExceptionReporter(ExceptionReporter reporter)
	{
		SecurityManager sm = System.getSecurityManager();
		if(sm != null)
		{
			sm.checkPermission(
				new ExceptionReporterPermission("exc.reporter"));
		}
		// Proceed to set the exception reporter
	}
	// ... Other methods of MyExceptionReporter
}
final class ExceptionReporterPermission extends BasicPermission
{
	public ExceptionReporterPermission(String permName)
	{
		super(permName);
	}
	// Even though the actions parameter is ignored,
	// this constructor has to be defined
	public ExceptionReporterPermission(String permName,
	String actions)
	{
		super(permName, actions);
	}
}
```

فایل سیاست به اعطای دو مجوز نیاز دارد: *exc.reporter ExceptionReporterPermission* و *loadlibrary.myLib RuntimePermission*. فایل سیاست زیر فرض می‌کند که منابع سابق، در دایرکتوری c:\package در یک سیستم مبتنی بر ویندوز قرار دارند.

```
grant codeBase "file:/c:/package"
{
      // For *nix, file:${user.home}/package/
      permission ExceptionReporterPermission "exc.reporter";
      permission java.lang.RuntimePermission "loadLibrary.myLib";
};
```

به‌طور پیش‌فرض، مجوزها نمی‌توانند برای پشتیبانی از عمل‌ها، توسط *Basic-Permission* تعریف شوند، اما عمل‌ها در صورت نیاز می‌توانند آزادانه در زیرکلاس *Exception-ReporterPermission* پیاده‌سازی گردند. *BasicPermission*، انتزاعی[۱۰۰] است، حتی اگر حاوی هیچ متد انتزاعی نباشد؛ تمام متدهایی را که از کلاس *Permission* توسعه می‌یابند، تعریف می‌کند. زیرکلاسی که به‌صورت سفارشی از کلاس *BasicPermission* تعریف شده‌است، باید دو سازنده را برای فراخوانی مناسب‌ترین سازنده‌ی ابرکلاس[۱۰۱] (یک یا دوآرگومانی) تعریف کند (زیرا ابرکلاس فاقد سازنده‌ی پیش‌فرض است). سازنده‌ی دوآرگومانی نیز، حتی در صورتی‌که یک مجوز پایه از آن استفاده نکند، یک عمل را می‌پذیرد. این رفتار، برای ساخت اشیای مجوز از فایل سیاست، لازم است. توجه داشته‌باشید که زیرکلاس تعریف‌شده‌ی سفارشی از کلاس *BasicPermission*، به‌عنوان *final* اعلان می‌گردد.

۳.۱۹.۲ کاربرد

اجرای کد جاوا، بدون تعریف مجوزهای سفارشی، که مجوزهای پیش‌فرض در آنها قابل استعمال نیستند، می‌تواند یک برنامه‌ی کاربردی را در معرض آسیب‌پذیری‌های افزایش امتیاز قرار دهد.

[۱۰۰] Abstract

[۱۰۱] Superclass

۲۰,۲ با استفاده از یک مدیر امنیت، یک جعبه‌شنی امن بسازید

مدیر امنیت، یک کلاس است که به برنامه‌های کاربردی اجازه می‌دهد تا یک سیاست امنیت را پیاده‌سازی کنند. به یک برنامه‌ی کاربردی اجازه می‌دهد تا پیش از انجام یک عمل احتمالاً غیرامن یا حساس، تعیین نماید که آن عمل چیست و آیا در تلاش است تا اجازه‌ی انجام آن را دهد یا خیر. برنامه‌ی کاربردی می‌تواند اجازه‌ی انجام آن عمل را بدهد و یا آن را ممنوع سازد. می‌توان یک مدیر امنیت را با هر کد جاوایی مرتبط نمود.

مدیر امنیت اَپلِت، تمام اپلت‌ها، به‌جز ضروری‌ترین امتیازات، را رد می‌کند. برای محافظت در برابر تغییر غیرعمد سیستم، نَشت اطلاعات، و جعل هویت کاربر، طراحی شده‌است. استفاده از مدیر امنیت، محدود به محافظت در سمت مشتری نیست. وب‌سرورها، مانند *Tomcat* و *WebSphere*، از این امکان برای ایزوله نمودن سِرولِت‌های تروجان و JSPهای[۱۰۲] مخرب و نیز محافظت از منابع سیستمی حساس از دسترسی غیرعمد، استفاده می‌کنند.

برنامه‌های کاربردی جاوا که از خط فرمان اجرا می‌شوند، می‌توانند با استفاده از یک پرچم خط فرمان، یک مدیر امنیت پیش‌فرض یا سفارشی تنظیم کنند. از سوی دیگر، نصب یک مدیر امنیت از طریق برنامه‌نویسی امکان‌پذیر است. این نوع نصب، به ساخت یک جعبه‌ی شنی پیش‌فرض، که اجازه‌ی انجام عمل‌های حساس را می‌دهد یا از آنها ممانعت به‌عمل می‌آورد (بر مبنای سیاست امنیتی مورد استفاده)، کمک می‌کند.

از پلتفرم Java 2 SE به بعد، *SecurityManager*، یک کلاس غیرانتزاعی به‌شمار می‌رود. در نتیجه، هیچ نیازمندی صریحی برای بازنویسی متدهای آن وجود ندارد. به‌منظور ساخت و استفاده از یک مدیر امنیت از طریق برنامه‌نویسی، کد باید مجوزهای زمان اجرا *createSecurityManager* (برای مقداردهی اولیه‌ی *SecurityManager*) و *setSecurityManager* (برای نصب آن) را داشته‌باشد. این مجوزها، تنها زمانی بررسی می‌شوند که یک مدیر امنیت، از قبل نصب شده‌باشد. این امر برای وضعیت‌هایی مفید است که در

[۱۰۲] Java Server Pages

آنها یک مدیر امنیت پیش‌فرض وجود داشته‌باشد، مانند روی یک میزبان مجازی، و میزبان‌های منفرد باید مجوزهای لازم برای بازنویسی مدیر امنیت پیش‌فرض را توسط نمونه‌ی سفارشی آن، رد نمایند.

مدیر امنیت، ارتباط نزدیکی با کلاس *AccessController* دارد. اولی به‌عنوان یک هاب برای کنترل دسترسی استفاده می‌شود، در حالی‌که دومی یک پیاده‌سازی حقیقی از الگوریتم کنترل دسترسی را فراهم می‌سازد. مدیر امنیت از موارد زیر پشتیبانی می‌کند:

- تعبیه‌ی سازگاری با گذشته: کد موروثی[۱۰۳]، اغلب شامل پیاده‌سازی‌های سفارشی از کلاس مدیر امنیت است، زیرا در اصل انتزاعی بود.
- تعریف سیاست‌های سفارشی: ساخت زیرکلاس مدیر امنیت، اجازه‌ی تعریف سیاست‌های امنیت سفارشی را می‌دهد (به‌عنوان نمونه، چندسطحی، ناهنجار یا خوب).

استفاده از *AccessController* در کد برنامه‌ی کاربردی توصیه می‌شود. این در حالی است که سفارشی‌سازی مدیر امنیت (از طریق زیرکلاس‌سازی) باید آخرین راه‌حل باشد و با احتیاط زیاد انجام پذیرد. علاوه بر این، یک مدیر امنیت سفارشی، مانند آن مدیر امنیتی که همواره زمان روز را پیش از فراخوانی بررسی‌های امنیتی استاندارد بررسی می‌کند، می‌تواند و باید هر زمان که مناسب بود، از الگوریتم فراهم‌شده توسط *AccessController* استفاده کند.

بسیاری از Java SE APIها، بررسی‌های مدیر امنیت را به‌صورت پیش‌فرض و قبل از انجام عملیات حساس، انجام می‌دهند. به‌عنوان نمونه، در صورتی‌که فراخواننده، مجوز خواندن یک فایل را در اختیار نداشته‌باشد، سازنده‌ی کلاس *java.io.FileInputStream*، استثنای *SecurityException* را ایجاد می‌نماید. چون *SecurityException* یک زیرکلاس از *RuntimeException* است، ممکن است اعلان برخی از روش‌های API (مانند آنهایی که متعلق به کلاس *java.io.FileReader* هستند)، فاقد یک عبارت برای ایجاد استثنایی باشند که *SecurityException* را فهرست می‌کند. از وابستگی به حضور یا عدم حضور بررسی‌های مدیر امنیت که در مستند روش API مشخص نشده‌اند، اجتناب نمایید.

---

[۱۰۳] Legacy code

1.20.2 نمونه‌کد ناسازگار (نصب از خط فرمان)

این مثال، در نصب هر مدیر امنیتی از خط فرمان شکست می‌خورد. در نتیجه، تمام مجوزها فعال می‌شوند و برنامه با آنها اجرا می‌شود. در واقع، هیچ مدیر امنیتی برای جلوگیری از اعمال شرورانه‌ای که ممکن است برنامه انجام دهد، وجود ندارد.

```
java LocalJavaApp
```

۲,۲۰,۲ راه‌حل سازگار (فایل سیاست پیش‌فرض)

هر برنامه‌ی جاوا می‌تواند تلاش کند یک مدیر امنیت را به‌صورت برنامه‌نویسی نصب کند. اگرچه، ممکن است مدیر امنیت فعال فعلی، این عمل را ممنوع کند. برنامه‌های کاربردی طراحی‌شده برای اجرای محلی، می‌توانند یک مدیر امنیت پیش‌فرض را با استفاده از یک پرچم در خط فرمان و در هنگام فراخوانی، مشخص کنند.

هنگامی‌که برنامه‌های کاربردی باید از نصب مدیرهای امنیت سفارشی با برنامه‌نویسی منع شوند، گزینه‌ی خط فرمان در اولویت بوده و نیاز است تا توسط سیاست امنیت پیش‌فرض، تحت تمام شرایط، پایدار بماند. این راه‌حل، مدیر امنیت پیش‌فرض را با استفاده از پرچم‌های خط فرمان مناسب نصب می‌کند. فایل سیاست امنیت، مجوزها را به عمل‌های مورد نظر برنامه‌ی کاربردی، اعطا می‌کند.

```
java -Djava.security.manager -Djava.security.policy=policyURL \LocalJavaApp
```

پرچم خط فرمان می‌تواند مدیر امنیت سفارشی را، که سیاست‌های آن به‌صورت سراسری اعمال شده‌اند، مشخص نماید. از پرچم *-Djava.security.manager* به‌صورت زیر استفاده کنید:

```
java -Djava.security.manager=my.security.CustomManager ...
```

اگر سیاست امنیت کنونی، که توسط مدیر امنیت کنونی اعمال شده‌است، جایگزینی‌ها را ممنوع سازد (با حذف RuntimePermission(“setSecurityManager”))، هر تلاشی برای فراخوانی *setSecurityManager()*، منجر به ایجاد استثنای *SecurityException* خواهد شد.

فایل سیاست امنیت پیش‌فرض *java.policy*، که در دایرکتوری /path/to/java.home/lib/security در سیستم‌های مشابه با UNIX و معادل آن در سیستم‌های مبتنی بر ویندوز یافت می‌شود، مجوزهای کمی، از قبیل خواندن ویژگی‌های سیستم، اتصال به[۱۰۴] پورت‌های غیرمجاز، و غیره، اعطا می‌نماید. ممکن است یک فایل سیاست مختص کاربر، در دایرکتوری home کاربر نیز قرار گیرد. اجتماع این فایل‌های سیاست، مجوزهای اعطاشده به یک برنامه را مشخص می‌نماید. فایل *java.security* می‌تواند مشخص کند که کدام فیل‌های سیاست مورد استفاده قرار گرفته‌اند. اگر هر یک از فایل‌های گستره‌ی سیستم *java.policy* یا *java.security* حذف شوند، هیچ مجوزی برای اجرای برنامه‌ی جاوا اعطا نمی‌شود.

۳,۲۰,۲راه‌حل سازگار (فایل سیاست سفارشی)

هنگام بازنویسی فایل سیاست عمومی جاوا با یک فایل سیاست سفارشی، از (==) به جای (=) استفاده کنید:

```
java -Djava.security.manager \
-Djava.security.policy==policyURL \
LocalJavaApp
```

۴,۲۰,۲راه‌حل سازگار (فایل‌های سیاست اضافی)

appletviewer، به‌صورت خودکار، یک مدیر امنیت را توسط فایل سیاست استاندارد نصب می‌کند. برای مشخص کردن فایل‌های سیاست اضافی، از پرچم *J*– استفاده کنید.

```
appletviewer -J-Djava.security.manager \
-J-Djava.security.policy==policyURL LocalJavaApp
```

توجه داشته‌باشید، هنگامی‌که ویژگی *policy.allowSystemProperty* در فایل ویژگی‌های امنیت (*java.security*) به *false* تنظیم گردد، فایل سیاست مشخص‌شده در آرگومان، نادیده گرفته می‌شود؛ مقدار پیش‌فرض این ویژگی، true است.

[۱۰۴] Binding to

۵,۲۰,۲نمونه‌کد ناسازگار (نصب با برنامه‌نویسی)

یک *SecurityManager* می‌تواند با استفاده از روش ایستای *System.setSecurity-Manager()* نیز، فعال شود. تنها یک *SecurityManager* در هر لحظه می‌تواند فعال باشد. این متد، *SecurityManager* فراهم‌شده در آرگومان را جایگزین *SecurityManager* فعال کنونی می‌کند. در صورت *null* بودن آرگومان، با هیچ *SecurityManager* جایگزین نمی‌شود.

این مثال، هر *SecurityManager* فعلی را غیرفعال می‌کند، اما *SecurityManager* دیگری به جای آن نصب نمی‌کند. در نتیجه، کد متعاقب با تمام مجوزهای فعال اجرا خواهد شد؛ هیچ محدودیتی روی هیچ عمل شرورانه‌ای که ممکن است برنامه انجام دهد، وجود نخواهد داشت.

```
try
{
	System.setSecurityManager(null);
}
catch (SecurityException se)
{
	// Cannot set security manager, log to file
}
```

یک مدیر امنیت فعال، که یک سیاست امنیت معقول را اعمال می‌کند، نمی‌گذارد سیستم آن را غیرفعال نماید و منجر به ایجاد استثنای *SecurityException* توسط این کد می‌شود.

۶,۲۰,۲راه‌حل سازگار (مدیر امنیت پیش‌فرض)

این مثال، مدیر امنیت پیش‌فرض را مقداردهی اولیه و تنظیم می‌کند.

```
try
{
	System.setSecurityManager(new SecurityManager());
}
catch (SecurityException se)
{
	// Cannot set security manager, log appropriately
}
```

۷.۲۰.۲راه‌حل سازگار (مدیر امنیت سفارشی)

این راه‌حل، چگونگی مقداردهی اولیه‌ی یک کلاس سفارشی *Security-Manager* را نشان می‌دهد که *CustomSecurityManager* را با فراخوانی سازنده‌ی آن توسط یک رمز عبور، فراخوانی کرده‌است. سپس، این مدیر امنیت سفارشی، به‌عنوان مدیر امنیت فعال نصب می‌شود.

```
char password[] = /* initialize */
try
{
	System.setSecurityManager(new CustomSecurityManager("password here"));
}
catch (SecurityException se)
{
	// Cannot set security manager, log appropriately
}
```

پس از اجرای این کد، APIهایی که بررسی‌های امنیتی را انجام می‌دهند، از مدیر امنیت سفارشی استفاده خواهند کرد. همان‌گونه که پیش از این اشاره شد، مدیرهای امنیت سفارشی باید تنها زمانی نصب شوند که مدیر امنیت فعلی، فاقد عملکردهای مورد نیاز باشد.

۸.۲۰.۲کاربرد

اساساً، امنیت جاوا به وجود یک مدیر امنیت بستگی دارد. در غیاب آن، عمل‌های حساس می‌توانند بدون محدودیت اجرا شوند.

تشخیص برنامه‌گونه‌ی حضور یا غیاب یک *SecurityManager* در زمان اجرا، امری سرراست است. تجزیه و تحلیل‌های ایستا می‌تواند به حضور یا غیاب کدی که اشاره کند که اگر اجرا می‌شد، تلاش می‌نمود یک *SecurityManager* را نصب نماید. ممکن است بررسی این‌که آیا *SecurityManager* به اندازه‌ی کافی زود نصب شده‌است یا خیر، یا این‌که آیا ویژگی‌های مطلوب را مشخص می‌کند، یا این‌که آیا نصب آن تضمین شده‌است، در برخی موارد خاص، امکان‌پذیر باشد، اما عموماً قابل تصمیم‌گیری نیست.

فراخوانی متد *setSecurityManager()* می‌تواند در محیط‌های کنترل‌شده، که در آنها می‌دانیم همواره، یک مدیر امنیت عمومی پیش‌فرضی از طریق خط فرمان نصب شده‌است، حذف شود. اِعمال این‌کار، مشکل است و اگر محیط به‌صورت نادرست پیکربندی شده‌باشد، می‌تواند منجر به آسیب‌پذیری‌هایی شود.

۲۱.۲ اجازه ندهید کد غیرقابل اعتماد از امتیازات روش‌های بازفراخوانی سوءاستفاده کند

بازفراخوانی‌ها[۱۰۵]، در هنگام وقوع یک رخداد جالب، وسیله‌ای را برای ثبت روشی، که باید فراخوانی (یا بازفراخوانی) شود، فراهم می‌آورند. جاوا، از بازفراخوانی‌ها برای رخدادهای اَپلِت‌ها و سِروِلِت‌ها، اخطارهای رخداد AWT و Swing، مانند کلیک‌های دکمه، خواندن‌ها و نوشتن‌های غیرهمزمان در فضای ذخیره‌سازی، و حتی *Runnable.run()*، که در آن یک نخ جدید به‌صورت خودکار روش *run()* مشخص‌شده را اجرا می‌کند، استفاده می‌نماید.

در جاوا، معمولاً بازفراخوانی‌ها با استفاده از واسط‌ها، پیاده‌سازی می‌شوند. ساختار کلی یک بازفراخوانی به‌صورت زیر است:

```
public interface CallBack {
	void callMethod();
}
class CallBackImpl implements CallBack {
	public void callMethod()
	{
		System.out.println("CallBack invoked");
	}
}
class CallBackAction {
	private CallBack callback;
	public CallBackAction(CallBack callback)
	{
		this.callback = callback;
	}
	public void perform()
	{
		callback.callMethod();
	}
}
class Client {
	public static void main(String[] args)
	{
		CallBackAction action=new CallBackAction(new CallBackImpl());
		// ...
		action.perform(); // Prints "CallBack invoked"
	}
}
```

[۱۰۵] Callbacks

روش‌های بازفراخوانی، اغلب بدون تغییر در امتیازات، فراخوانی می‌شوند. در واقع، ممکن است در مفهومی اجرا شوند که امتیازات بیشتری نسبت به مفهومی دارد که در آن اعلان شده‌اند. اگر این روش‌های بازفراخوانی، داده‌ها را از کد غیرقابل اعتماد بپذیرند، ممکن است افزایش امتیاز رخ دهد.

۱.۲۱.۲ نمونه کد ناسازگار

این مثال، از یک کلاس *UserLookupCallBack* استفاده می‌کند که واسط *CallBack* را با استفاده از ID، برای جست‌وجوی نام یک کاربر پیاده‌سازی می‌کند. این کد جست‌وجو فرض می‌کند که این اطلاعات در فایل /etc/passwd قرار دارند و به امتیاز عالی[۱۰۶] برای باز کردن آن نیاز است. در نتیجه، کلاس *Client*، تمام بازفراخوانی‌ها با امتیازات عالی را فراخوانی می‌نماید (با یک بلاک *doPrivileged*).

```
public interface CallBack
{
	void callMethod();
}
class UserLookupCallBack implements CallBack
{
	private int uid;
	private String name;
	public UserLookupCallBack(int uid)
	{
		this.uid = uid;
	}
	public String getName()
	{
		return name;
	}
	public void callMethod()
	{
		try (InputStream fis = new FileInputStream("/etc/passwd"))
		{
			// Look up uid & assign to name
		}
		catch (IOException x)
		{
			name = null;
		}
	}
}
```

[۱۰۶] Elevated privilege

```
final class CallBackAction
{
	private CallBack callback;
	public CallBackAction(CallBack callback)
	{
		this.callback = callback;
	}
	public void perform()
	{
		AccessController.doPrivileged(
			new PrivilegedAction<Void>()
			{
				public Void run()
				{
					callback.callMethod();
					return null;
				}
			});
	}
}
```

این کد می‌تواند به‌صورت امن، توسط مشتری و به شکل زیر استفاده شود:

```
public static void main(String[] args)
{
	int uid = Integer.parseInt(args[0]);
	CallBack callBack = new UserLookupCallBack(uid);
	CallBackAction action = new CallBackAction(callBack);
	// ...
	action.perform(); // Looks up user name
	System.out.println("User " + uid + " is named " +
	callBack.getName());
}
```

با این‌وجود، یک مهاجم می‌تواند از *CallBackAction* برای اجرای کد مخرب با امتیازات عالی، با ثبت یک نمونه‌ی *MaliciousCallBack*، استفاده کند:

```
class MaliciousCallBack implements CallBack
{
	public void callMethod()
	{
		// Code here gets executed with elevated privileges
	}
}
// Client code
public static void main(String[] args)
{
	CallBack callBack = new MaliciousCallBack();
	CallBackAction action = new CallBackAction(callBack);
	action.perform(); // Executes malicious code
}
```

۲,۲۱,۲ راه‌حل سازگار (بازفراخوانی- بلاک محلی *doPrivileged*)

این راه حل، فراخوانی *doPrivileged()* را به خارج از کد *CallBackAction* و داخل خود بازفراخوان، می‌برد. این کد، رفتاری مانند قبل دارد، اما مهاجم، دیگر نمی‌تواند کد بازفراخوانی مخرب را با امتیازات افزایش‌یافته اجرا نماید. حتی اگر مهاجم بتواند یک نمونه‌ی بازفراخوانی مخرب را با استفاده از سازنده‌ی کلاس *CallBackAction* انتقال دهد، کد با امتیازات افزایش‌یافته، اجرا نمی‌شود، زیرا نمونه‌ی مخرب باید حاوی یک بلاک *doPrivileged* باشد که نتواند امتیازاتی مشابه با یک کد قابل اعتماد داشته باشد. علاوه بر این، کلاس *CallBackAction* نمی‌تواند برای بازنویسی روش *perform()* زیرکلاس‌سازی شود، زیرا به‌عنوان نهایی اعلان شده‌است.

```
public interface CallBack {
	void callMethod();
}
class UserLookupCallBack implements CallBack {
	private int uid;
	private String name;
	public UserLookupCallBack(int uid)
	{
		this.uid = uid;
	}
	public String getName()
	{
		return name;
	}
	public final void callMethod()
	{
		AccessController.doPrivileged(
			new PrivilegedAction<Void>()
			{
				public Void run()
				{
					try (InputStream fis =
					new FileInputStream("/etc/passwd"))
					{
						// Look up userid and assign to
						// UserLookupCallBack.this.name
					}
					catch (IOException x)
					{
						UserLookupCallBack.this.name = null;
					}
					return null;
				}
			});
	}
}
final class CallBackAction {
	private CallBack callback;
	public CallBackAction(CallBack callback)
	{
		this.callback = callback;
	}
	public void perform()
	{
		callback.callMethod();
	}
}
```

۳,۲۱,۲راه‌حل سازگار

این راه‌حل، کلاس *UserLookupCallBack* را به‌صورت نهایی اعلان می‌کند تا جلوی بازنویسی *callMethod()* را بگیرد.

```
final class UserLookupCallBack implements CallBack
{
      // ...
}
// Remaining code is unchanged
```

۴,۲۱,۲کاربرد

افشای روش‌های حساس از طریق بازفراخوانی، می‌تواند منجر به سوءاستفاده از امتیازات و اجرای کد دلخواه شود.

فصل سوم: برنامه‌نویسی تدافعی

برنامه‌نویسی تدافعی، یک برنامه‌نویسی به‌دقت محافظت‌شده است که به شما کمک می‌کند تا نرم‌افزاری قابل اعتماد را، با طراحی هر مولفه به‌گونه‌ای که تا حد ممکن از خودش محافظت کند، بسازید. راهنماهای این بخش، به حوزه‌های زبان جاوا، که می‌توانند به محدود کردن اثر یک خطا یا بازیابی از یک خطا کمک کنند، می‌پردازند.

سازوکارهای زبان جاوا باید برای محدودسازی حوزه[۱۰۷]، طول عمر، و قابلیت دسترسی منابع برنامه، استفاده شوند. همچنین، حاشیه‌نگاری‌های جاوا[۱۰۸] می‌توانند برای مستندسازی و کمک به خوانایی و نگهداری برنامه، استفاده گردند. برنامه‌نویسان جاوا باید از رفتارهای ضمنی باخبر باشند و از فرضیات غیرقابل ضمانت[۱۰۹] در مورد این‌که سیستم چگونه رفتار می‌کند، بپرهیزند.

یک اصل کلی خوب برای برنامه‌نویسی تدافعی، سادگی است. فهم و نگهداری یک سیستم پیچیده و نیز درست بودن آن، دشوار است. اگر پیاده‌سازی یک سازنده پیچیده باشد، طراحی مجدد یا بازسازی[۱۱۰] آن را مدنظر داشته‌باشید.

در نهایت، برنامه باید تا حد ممکن به صورت مقاوم[۱۱۱] طراحی شود. هرجاکه امکان دارد، برنامه باید به سیستم زمان اجرای جاوا، با محدود کردن منابعی که استفاده می‌کند و نیز رهاسازی منابع به‌دست‌آمده که دیگر مورد نیاز نیستند، کمک کند. مجدداً، این امر می‌تواند با محدودسازی طول عمر، دسترس‌پذیری اشیا و سایر ساختارهای برنامه‌نویسی، به‌دست آید. نمی‌توان تمام احتمالات را پیش‌بینی نمود. بنابراین، باید راهبردی برای فراهم نمودن یک خروج مطبوع به عنوان آخرین راه‌حل، توسعه داد.

---

[۱۰۷] Scope
[۱۰۸] Java annotations
[۱۰۹] Unwarranted
[۱۱۰] Refactoring
[۱۱۱] Robust

1.3 حوزه‌ی متغیرها را کمینه نمایید

کمینه‌سازی حوزه، به توسعه‌دهندگان کمک می‌کند تا از خطاهای برنامه‌نویسی رایج اجتناب نمایند، خوانایی کد را با اتصال اعلان و استفاده‌ی واقعی از یک متغیر بهبود دهند، و قابلیت نگهداری را بهتر کنند، زیرا متغیرهایی که مورد استفاده قرار نگرفته‌اند، ساده‌تر شناسایی و حذف می‌شوند. همچنین، می‌تواند به اشیا اجازه دهد تا با استفاده از زباله‌روب، سریع‌تر بازیابی شوند.

1.1.3 نمونه‌کد ناسازگار

این مثال، متغیری را نشان می‌دهد که خارج از حلقه‌ی *for* اعلان شده‌است.

```
public class Scope
{
	public static void main(String[] args)
	{
		int i = 0;
		for (i = 0; i < 10; i++)
		{
			// Do operations
		}
	}
}
```

این کد، ناسازگار است، زیرا با وجود این‌که متغیر *i* سهوا خارج از حلقه‌ی *for* استفاده شده، اما در حوزه‌ی متد، اعلان شده‌است. یکی از سناریوهایی که در آن متغیر *i* به اعلان در حوزه‌ی متد نیاز دارد، زمانی است که حلقه، حاوی یک عبارت *break* بوده و مقدار *i* باید پس از خاتمه‌ی حلقه، بررسی گردد.

۲.۱.۳ راه‌حل سازگار

هر جاکه ممکن بود، حوزه‌ی متغیرها را کمینه کنید. به‌عنوان نمونه، اندیس‌های حلقه را درون عبارت *for* اعلان نمایید:

```
public class Scope {
	public static void main(String[] args) {
		for (int i = 0; i < 10; i++) { // Contains declaration
			// Do operations
		}
	}
}
```

۳,۱,۳ نمونه‌کد ناسازگار

این مثال، یک متغیر *count* را نشان می‌دهد که خارج از محدوده‌ی متد *counter()* اعلان شده‌است. اگرچه، متغیر، خارج از محدوده‌ی متد *counter()*، مورد استفاده قرار نگرفته‌است.

```
public class Foo
{
	private int count;
	private static final int MAX_COUNT = 10;
	public void counter()
	{
		count = 0;
		while (condition())
		{
			/* ... */
			if (count++ > MAX_COUNT)
			{
				return;
			}
		}
	}
	private boolean condition()
	{
		/* ... */
	}
	// No other method references count
	// but several other methods reference MAX_COUNT
}
```

قابلیت استفاده‌ی مجدد از این روش، کاهش یافته‌است، زیرا اگر روش به کلاس دیگری کپی شود، متغیر *count* نیز باید مجددا در مفهوم جدید، تعریف گردد. علاوه بر این، تحلیل‌پذیری متد *counter* کاهش می‌یابد، زیرا کل تحلیل جریان داده‌ای برنامه برای تعیین مقادیر ممکن *count*، ضروری خواهد بود.

۴,۱,۳ راه‌حل سازگار

در این راه‌حل، فیلد *count* به‌صورت محلی برای متد *counter()* اعلان شده‌است:

```
public class Foo
{
	private static final int MAX_COUNT = 10;
	public void counter()
	{
		int count = 0;
		while (condition())
		{
			/* ... */
			if (count++ > MAX_COUNT)
			{
				return;
			}
		}
	}
	private boolean condition()
	{
		/* ... */
	}
	// No other method references count
	// but several other methods reference MAX_COUNT
}
```

۵,۱,۳ کاربرد

تشخیص متغیرهای محلی، که در حوزه‌ی بزرگتری از نیاز کد اعلان می‌شوند، آسان است و می‌تواند احتمال مثبت‌های غلط[۱۱۲] را از بین ببرد. تشخیص چندین عبارت، که از متغیر اندیس یکسانی استفاده می‌کنند، ساده است؛ فقط در موارد غیرمعمولی، که مقدار متغیر اندیس به‌صورتی درنظر گرفته شده‌باشد که بین حلقه‌ها ادامه یابد، مثبت غلط تولید می‌کند.

[۱۱۲] False positives

۲,۳ حوزه‌ی حاشیه‌نگاری *@SuppressWarnings* را کمینه نمایید

هنگامی‌که کامپایلر، مسائل احتمالی نوع-ایمن[۱۱۳] به وجود آمده ناشی از ترکیب انواع خام با کد عمومی[۱۱۴] را تشخیص می‌دهد، هشدارهای *unchecked warnings*، شامل *unchecked cast warnings*، *unchecked method invocation warnings*، *unchecked generic array creation* warnings و *unchecked conversion warnings* را می‌فرستد. استفاده از حاشیه‌نگاری *@SuppressWarnings("unchecked")* برای جلوگیری از[۱۱۵] هشدارهای بررسی‌نشده، تنها هنگامی‌که ایمن بودن کد حذف‌کننده‌ی هشدار تضمین شود، مجاز است. یک مورد استفاده‌ی رایج، ترکیب کد قدیمی با کد مشتری جدید است. خطرات جدی نادیده گرفتن هشدارهای بررسی‌نشده، به‌طور گسترده، در *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012]، (OBJ03-J. در کد جدید، انواع خام عمومی را با غیرعمومی ترکیب نکنید)، بررسی شده‌است.

حاشیه‌نگاری *@SuppressWarnings* می‌تواند در اعلان متغیرها، متدها، و کل یک کلاس استفاده شود. با این‌حال، باریک کردن حوزه‌ی آن، امری مهم تلقی می‌شود تا تنها، از هشدارهایی که در حوزه‌ی باریک‌تر رخ می‌دهند، جلوگیری به‌عمل آید.

1.2.3 نمونه‌کد ناسازگار

در این نمونه‌کد، حوزه‌ی حاشیه‌نگاری *@SuppressWarnings*، کل کلاس را دربر می‌گیرد:

```
@SuppressWarnings("unchecked")
class Legacy
{
	Set s = new HashSet();
	public final void doLogic(int a, char c)
	{
		s.add(a);
		s.add(c); // Type-unsafe operation, ignored
	}
}
```

[۱۱۳] Type-safety

[۱۱۴] Generic code

[۱۱۵] Suppress

این کد، خطرناک است، زیرا از تمام هشدارهای بررسی‌نشده درون کلاس، جلوگیری شده‌است. همچنین، می‌تواند در زمان اجرا، منجر به استثنای *ClassCastException* شود.

۲.۲.۳ راه‌حل سازگار

حوزه‌ی حاشیه‌نگاری *@SuppressWarnings* را به نزدیک‌ترین کدی که یک هشدار تولید می‌کند، محدود نمایید. در این حالت، می‌تواند در اعلان *Set* استفاده شود:

```
class Legacy
{
	@SuppressWarnings("unchecked")
	Set s = new HashSet();
	public final void doLogic(int a, char c)
	{
		s.add(a); // Produces unchecked warning
		s.add(c); // Produces unchecked warning
	}
}
```

۳.۲.۳ نمونه‌کد ناسازگار (*ArrayList*)

این نمونه‌کد، ناشی از یک پیاده‌سازی قدیمی *java.util .ArrayList* است:

```
@SuppressWarnings("unchecked")
public <T> T[] toArray(T[] a)
{
	if (a.length < size)
	{
		// Produces unchecked warning
		return (T[]) Arrays.copyOf(elements, size, a.getClass());
	}
	// ...
}
```

هنگامی‌که کلاس کامپایل شود، یک هشدار پخش بدون کنترل[۱۱۶] منتشر می‌کند .از این هشدار نمی‌توان فقط برای عبارت *return*، توقیف شود، زیرا یک اعلان نیست. در نتیجه، برنامه‌نویس باید هشدارها را برای کل متد توقیف کند. وقتی وظایفی که عملیات نوع-غیرامنی را انجام می‌دهند، بعداً به متد اضافه شوند، این امر می‌تواند مشکلاتی به‌وجود آورد.

[۱۱۶] Unchecked cast warning

```
// Unchecked cast warning
ArrayList.java:305: warning: [unchecked] unchecked cast found :
Object[], required: T[]
return (T[]) Arrays.copyOf(elements, size, a.getClass());
```

۴،۲،۳ راه‌حل سازگار (*ArrayList*)

وقتی استفاده از حاشیه‌نگاری *@SuppressWarnings* در یک حوزه‌ی مناسب، دشوار است (مانند نمونه‌کد ناسازگار پیشین)، یک متغیر جدید تعریف کنید تا مقدار بازگشتی را نگه دارد و آن را با حاشیه‌نگاری *@SuppressWarnings*، آراسته کنید.

```
// ...
@SuppressWarnings("unchecked")
T[] result = (T[]) Arrays.copyOf(elements, size, a.getClass());
return result;
// ...
```

۵،۲،۳ کاربرد

شکست در کاهش حوزه‌ی حاشیه‌نگاری *@SuppressWarnings*، می‌تواند منجر به استثنائات زمان اجرا شود و تضمین‌های نوع-ایمن را بشکند. این قانون می‌تواند به‌صورت ایستا و با عمومیت کامل، اعمال شود. با این‌وجود، تجزیه و تحلیل ایستا می‌تواند در برخی موارد خاص استفاده شود.

۳,۳ قابلیت دسترسی کلاس‌ها و اعضایشان را کمینه کنید

کلاس‌ها و اعضای کلاس جاوا (کلاس‌ها، واسط‌ها، فیلدها و متدها) کنترل دسترسی می‌شوند. دسترسی، توسط یک تعیین‌گر سطح دسترسی[۱۱۷] (*public*، *protected* یا *private*)، یا با عدم حضور یک تعیین‌گر سطح دسترسی (دسترسی پیش‌فرض، *package-private access* نیز نام دارد) نشان داده می‌شود.

جدول ۳-۱، دید ساده‌شده‌ای از قوانین کنترل دسترسی را نشان می‌دهد. یک *x* نشان می‌دهد که دسترسی خاص، از درون دامنه، مجاز است. به‌عنوان نمونه، یک *x* در ستون کلاس به معنای قابل دسترس بودن عضو کلاس برای کد حاضر در همان کلاسی است که در آن اعلان شده‌است. به‌طور مشابه، ستون بسته نشان می‌دهد که آن عضوی از کلاس (یا زیرکلاس) که در آن بسته تعریف شده، به‌شرطی‌که کلاس (یا زیرکلاس) توسط بارگذار کلاسی که کلاس حاوی عضو را بارگذاری نموده‌بود، بارگذاری شود، قابل دسترسی است. شرایط مشابه با بارگذار کلاس، تنها به دسترسی عضو *package-private* اعمال می‌گردد.

جدول ۳-۱: کلاس‌های شامل عضو

تعیین‌گر دسترسی	کلاس	بسته	زیرکلاس	دنیا
Private	✓			
None	✓	✓	✓	
Protected	✓	✓	✓	
Public	✓	✓	✓	✓

باید کمترین دسترسی ممکن به کلاس‌ها و اعضای کلاس داده شود تا کد مخرب، حداقل فرصت را برای به خطر انداختن امنیت داشته‌باشد. تا حد ممکن، کلاس‌ها باید از افشای متدهایی که حاوی کد حساس هستند یا آن را فراخوانی می‌کنند، توسط واسط اجتناب نمایند. واسط‌ها، تنها متدهایی را که در دسترس عموم هستند، می‌پذیرند. چنین متدهایی، بخشی از API عمومی کلاس هستند. یک استثنا برای این موضوع، پیاده‌سازی یک واسط *unmodifiable* است که دید غیرقابل تغییر عمومی را از یک شی قابل تغییر، افشا می‌کند (OBJ04-J. وظیفه‌ی کپی کردن را برای کلاس‌های قابل تغییر فراهم کنید تا اجازه‌ی ارسال نمونه‌ها به کد غیرقابل اعتماد را بدهد). توجه داشته‌باشید که حتی اگر یک کلاس غیرنهایی،

[۱۱۷] Access modifier

به‌صورت پیش‌فرض قابل رؤیت باشد، در صورتی‌که حاوی متدهای عمومی بوده، می‌تواند مستعد سوءاستفاده باشد. متدهایی که تمام بررسی‌های امنیتی لازم را انجام می‌دهند و تمام ورودی‌ها را پاک‌سازی می‌نمایند، ممکن است از طریق واسط‌ها، در معرض خطر قرار گیرند.

قابلیت دسترسی محافظت‌شده، برای کلاس‌های غیر تودرتو، مجاز نیست، اما ممکن است کلاس‌های تودرتو، به‌صورت محافظت‌شده اعلان شوند. فیلدهای کلاس‌ها عمومی غیرنهایی، باید به‌ندرت به‌صورت محافظت‌شده اعلان شوند؛ کد غیرقابل اعتماد موجود در بسته‌ی دیگر، می‌تواند زیرکلاسی از کلاس باشد و به عضو، دسترسی یابد. علاوه بر این، اعضای محافظت‌شده، بخشی از API کلاس هستند و در نتیجه، به پشتیبانی مداوم نیاز دارند. هنگامی‌که این قانون دنبال شود، اعلان فیلدها به‌عنوان محافظت‌شده، غیرضروری می‌گردد (OBJ01-J. اعضای داده‌ای را به‌عنوان خصوصی اعلان کنید و متدهای پنهان‌ساز قابل دسترسی را فراهم نمایید).

اگر یک کلاس، واسط، متد، یا فیلد، بخشی از یک API منتشرشده، مانند یک نقطه‌ی انتهایی سرویس وب باشد، می‌تواند به‌صورت عمومی اعلان شود. کلاس‌ها و اعضای دیگر باید به‌صورت *package-private* یا خصوصی اعلان شوند. به‌عنوان نمونه، کلاس‌های غیر امنیتی حیاتی، تشویق می‌شوند تا تولیدگران ایستای عمومی را، به‌منظور پیاده‌سازی کنترل نمونه با یک سازنده‌ی خصوصی، فراهم سازند.

1.3.3 نمونه‌کد ناسازگار (کلاس عمومی)

این کد، کلاسی تعریف می‌کند که برای یک سیستم داخلی محسوب می‌شود و بخشی از هیچ API عمومی نیست. با این‌وجود، این کلاس به‌عنوان عمومی اعلان شده‌است.

```
public final class Point
{
	private final int x;
	private final int y;
	public Point(int x, int y)
	{
		this.x = x;
		this.y = y;
	}
	public void getPoint()
	{
		System.out.println("(" + x + "," + y + ")");
	}
}
```

اگرچه، این مثال با OBJ01-J مطابقت دارد، اما کد غیرقابل اعتماد می‌تواند *Point* را مقداردهی اولیه کند و متد عمومی *getPoint()* را فراخوانی نماید تا مختصات را به‌دست آورد.

۲.۳.۳ راه‌حل سازگار (کلاس‌های پایانی با متدهای عمومی)

این راه‌حل، کلاس *Point* را مطابق با وضعیت آن، که بخشی از هیچ API عمومی نیست، به‌عنوان *package-private* اعلان می‌کند:

```
final class Point
{
	private final int x;
	private final int y;
	Point(int x, int y)
	{
		this.x = x;
		this.y = y;
	}
	public void getPoint()
	{
		System.out.println("(" + x + "," + y + ")");
	}
}
```

یک کلاس سطح بالا، مانند *Point*، نمی‌تواند به‌صورت خصوصی اعلان شود. قابلیت دسترسی *package-private*، قابل قبول است و از حملات درج بسته[۱۱۸] اجتناب می‌کند (ENV01-J. تمام کدهای حساس به امنیت را در یک JAR منفرد قرار دهید و آن را امضا و مُهر نمایید). یک حمله‌ی درج بسته هنگامی رخ می‌دهد که در زمان اجرا، هر عضو محافظت‌شده یا *package-private* از یک کلاس، بتواند مستقیما توسط کلاسی که به‌صورت مخرب به همان بسته اضافه شده‌است، فراخوانی شود. با این‌وجود، انجام این حمله در عمل مشکل است، زیرا علاوه بر نیازمندی‌های نفوذپذیری[۱۱۹] به بسته، هدف و کلاس غیرقابل اعتماد باید توسط همان بارگذار کلاس، بارگذاری شوند. معمولاً کد غیرقابل اعتماد از چنین سطوح دسترسی، محروم است.

---

[۱۱۸] Package insertion attack

[۱۱۹] Infiltrating

از آنجایی‌که کلاس، پایانی است، متد *getPoint()* می‌تواند به‌صورت عمومی اعلان شود. یک زیرکلاس عمومی که این قانون را نقض می‌کند، نمی‌تواند متد را بازنویسی نماید و آن را در معرض کد غیرقابل اعتماد قرار دهد. بنابراین، قابلیت دسترسی آن نامربوط است. برای کلاس‌های غیرپایانی، کاهش قابلیت دسترسی متدها به خصوصی یا *package-private*، تهدید را برطرف می‌سازد.

۳،۳،۳ راه‌حل سازگار (کلاس‌های غیرپایانی با متدهای غیرعمومی)

این راه‌حل، کلاس *Point* و متد *getPoint()* آن را به‌صورت *package-private* اعلان می‌کند، که به کلاس *Point* اجازه می‌دهد، غیرپایانی باشد. همچنین، این انکان را برای متد *getPoint()* فراهم می‌آورد تا توسط کلاس موجود درون همان بسته، فراخوانی شده و توسط یک بار گذار مرسوم کلاس، بارگذاری شود:

```
class Point
{
	private final int x;
	private final int y;
	Point(int x, int y)
	{
		this.x = x;
		this.y = y;
	}
	void getPoint()
	{
		System.out.println("(" + x + "," + y + ")");
	}
}
```

۴،۳،۳ نمونه‌کد ناسازگار (کلاس عمومی با متد عمومی ایستا)

این نمونه‌کد، مجدداً کلاسی را تعریف می‌کند که برای سیستم، داخلی محسوب می‌شود و بخشی از هیچ API عمومی نیست. با این‌وجود، این کلاس، به‌صورت عمومی اعلان شده‌است.

```
public final class Point {
	private static final int x = 1;
	private static final int y = 2;
	private Point(int x, int y) {}
	public static void getPoint() {
		System.out.println("(" + x + "," + y + ")");
	}
}
```

این مثال، با OBJ01-J نیز همخوانی دارد. کد غیرقابل اعتماد می‌تواند به *Point* دسترسی داشته‌باشد و *getPoint()* را، که ایستای عمومی است، فراخوانی نماید تا مختصات پیش‌فرض را به‌دست آورد. تلاش برای پیاده‌سازی کنترل نمونه از طریق یک سازنده‌ی خصوصی، بیهوده است، زیرا متد ایستای عمومی، محتوای کلاس داخلی را در معرض خطر قرار می‌دهد.

۵،۳،۳ راه‌حل سازگار (کلاس *Package-Private*)

این راه‌حل، قابلیت دسترسی کلاس به *package-private* را کاهش می‌دهد. دسترسی به متد *getPoint()*، به کلاس‌هایی که درون همان بسته قرار دارند، محدود است. از فراخوانی *getPoint()* و به‌دست آوردن مختصات از طریق کد غیرقابل اعتماد، جلوگیری می‌شود.

```
final class Point
{
	private static final int x = 1;
	private static final int y = 2;
	private Point(int x, int y) {}
	public static void getPoint()
	{
		System.out.println("(" + x + "," + y + ")");
	}
}
```

۶،۳،۳ کاربرد

اعطای دسترسی بیش از اندازه، منجر به نقض کپسوله‌سازی می‌شود و امنیت برنامه‌های کاربردی جاوا را تضعیف می‌نماید.

برای هر تکه کد داده‌شده، کمترین قابلیت دسترسی برای هر کلاس و عضو، می‌تواند محاسبه شود تا از معرفی خطاهای کامپایل اجتناب گردد. یکی از محدودیت‌ها این است که ممکن است نتیجه‌ی این محاسبات، فاقد هر گونه تطابقی با آن‌چه که قصد برنامه‌نویس به هنگام نوشتن کد بوده‌است، باشد. به‌عنوان نمونه، اعضای استفاده‌نشده می‌توانند به‌وضوح به‌عنوان خصوصی اعلان شوند. با این‌حال، چنین اعضایی می‌توانند تنها به‌دلیل این‌که بدنه‌ی خاصی از کد (که به‌صورت تصادفی مورد بررسی قرار گرفته‌است) فاقد ارجاعاتی به اعضا باشد، مورد استفاده قرار نگیرند. با این‌وجود، این محاسبات می‌توانند نقطه‌ی آغازین

مفیدی برای برنامه‌نویسی که قصد دارد قابلیت دسترسی کلاس‌ها و اعضای آنها را کمینه نماید، فراهم آورند.

۴,۳ ایمنی نخ را مستند کنید و در جای مناسب، از حاشیه‌نگاری استفاده نمایید

امکان حاشیه‌نگاری زبان جاوا، برای مستندسازی اهداف طراحی، سودمند است. حاشیه‌نگاری کد، سازوکاری برای مرتبط کردن فراداده‌ها با یک عنصر برنامه و در دسترس قرار دادن آن برای کامپایلر، تجزیه تحلیلگرها، اشکال‌زداها، یا JVM، برای آزمایشات است. چندین حاشیه‌نگاری برای مستندسازی ایمنی نخ و یا فقدان آن وجود دارد.

۱,۴,۳ به‌دست آوردن حاشیه‌نگاری‌های همروندی

دو مجموعه از حاشیه‌نگاری‌های همروندی به‌صورت رایگان و برای استفاده در هر کدی، در دسترس و مجاز[۱۲۰] هستند: اولین مجموعه، از چهار حاشیه‌نگاری تشکیل شده‌است که می‌توان از آدرس http://jcip.net دانلود نمود. حاشیه‌نگاری‌های JCIP، تحت Creative Commons Attribution License منتشر شده‌اند؛ دومین مورد، که مجموعه‌ی بزرگتری از حاشیه‌نگاری‌های همروندی است، از طریق *SureLogic* در دسترس است و توسط آن پشتیبانی می‌شود. این حاشیه‌نگاری‌ها، تحت نسخه‌ی دوم Apache Software License منتشر شده‌اند و از آدرس www.surelogic.com قابل دانلود هستند. حاشیه‌نگاری‌ها را می‌توان با ابزار *SureLogic Jsure* وارسی نمود. در چنین شرایطی، حتی در صورت در دسترس نبودن ابزار، برای مستندسازی کد، سودمند خواهند بود. این حاشیه‌نگاری‌ها، شامل حاشیه‌نگاری‌های JCIP نیز هستند، زیرا توسط ابزار *JSure* پشتیبانی می‌شوند (*JSure*، استفاده از فایل JCIP JAR را نیز پوشش می‌دهد).

برای استفاده از حاشیه‌نگاری‌ها، یک یا هر دو فایل JAR را، که به آنها شاره شد، دانلود کنید و در مسیر *build* کد، اضافه نمایید. نحوه‌ی استفاده از این حاشیه‌نگاری‌ها در راستای مستندسازی ایمنی نخ، در ادامه‌ی این فصل توصیف شده‌است.

---

[۱۲۰] Licensed

۲.۴.۳ ایمنی نخ موردنظر را مستند کنید

JCIP، سه حاشیه‌نگاری سطح کلاس را فراهم می‌کند تا هدف طراحی برنامه‌نویس را، با درنظر داشتن ایمنی نخ، توصیف نماید.

حاشیه‌نگاری *@ThreadSafe* به یک کلاس اعمال می‌شود تا مشخص نماید که نخ، ایمن است. در واقع، هیچ دنباله‌ای از دسترسی‌ها (خواندن و نوشتن‌های فیلدهای عمومی، فراخوانی‌های متدهای عمومی) نمی‌تواند شی را در یک وضعیت ناپایدار، بدون در نظر گرفتن تعویض[۱۲۱] این دسترسی‌ها توسط زمان اجرا یا هر همگام‌سازی خارجی یا هماهنگ‌سازی در بخش فراخوانی‌کننده، رها نماید. به‌عنوان نمونه، کلاس *Aircraft* زیر مشخص می‌کند که نخ، مانند بخشی از مستند سیاست قفل‌گذاری، ایمن است. این کلاس، با استفاده از قفلی که مجددا ورود می‌یابد، از فیلدهای x و y محافظت می‌کند.

```
@ThreadSafe
@Region("private AircraftState")
@RegionLock("StateLock is stateLock protects AircraftState")
public final class Aircraft
{
	private final Lock stateLock = new ReentrantLock();
	// ...
	@InRegion("AircraftState")
	private long x, y;
	// ...
	public void setPosition(long x, long y)
	{
		stateLock.lock();
		try
		{
			this.x = x;
			this.y = y;
		}
		finally
		{
			stateLock.unlock();
		}
	}
	// ...
}
```

[۱۲۱] Interleaving

حاشیه‌نگاری‌های *@Region* و *@RegionLock*، سیاست قفل‌گذاری را مستند می‌کنند که بر اساس آن وعده‌ی ایمنی نخ، پیش‌بینی می‌شود.

حتی هنگامی‌که یک یا چند حاشیه‌نگاری *@RegionLock* یا *@GuardedBy* برای مستندسازی سیاست قفل‌گذاری یک کلاس استفاده می‌شوند، حاشیه‌نگاری *@ThreadSafe*، راهی بصری برای بازدیدکنندگان فراهم می‌سازد تا یاد بگیرند که کلاس، از نوع ایمنی نخ است.

حاشیه‌نگاری *@Immutable* به کلاس‌های *immutable* اعمال می‌شود. اشیای غیرقابل تغییر، ذاتاً از نوع ایمنی نخ هستند. به‌محض این‌که به‌طور کامل ساخته شدند، می‌توان آنها را از طریق یک ارجاع منتشر کرد و به‌صورت ایمن، بین چندین نخ به اشتراک گذاشت. مثال زیر، یک کلاس *Point* غیرقابل تغییر را نشان می‌دهد:

```
@Immutable
public final class Point
{
	private final int f_x;
	private final int f_y;
	public Point(int x, int y)
	{
		f_x = x;
		f_y = y;
	}
	public int getX()
	{
		return f_x;
	}
	public int getY()
	{
		return f_y;
	}
}
```

حاشیه‌نگاری *@NotThreadSafe*، به کلاس‌هایی اعمال می‌شود که از نوع ایمنی نخ نیستند. بسیاری از کلاس‌ها در استناد به این‌که آیا برای استفاده‌ی چندنخی، ایمن هستند، شکست می‌خورند. در نتیجه، یک برنامه‌نویس هیچ راه ندارد تا تعیین نماید که آیا کلاس، از نوع ایمنی نخ است یا خیر. این حاشیه‌نگاری، نشانه‌ی واضحی از فقدان ایمنی نخ را برای کلاس فراهم می‌کند. به‌عنوان نمونه، اغلب پیاده‌سازی‌های

مجموعه‌ای که در *java.util* ارائه شده‌اند، از نوع ایمنی نخ نیستند. کلاس *java.util.ArrayList* می‌تواند آن را به‌صورت زیر مستند کند:

```
package java.util.ArrayList;
@NotThreadSafe
public class ArrayList<E> extends ...
{
      // ...
}
```

۳.۴.۳ مستندسازی سیاست‌های قفل‌گذاری

مستندسازی تمام قفل‌ها، که برای حفاظت از وضعیت اشتراکی استفاد می‌شوند، امری مهم به‌حساب می‌آید. برای این‌منظور، *JCIP*، حاشیه‌نگاری *@GuardedBy* و *SureLogic*، حاشیه‌نگاری *@RegionLock* را فراهم می‌کنند. فیلد یا متدی که حاشیه‌نگاری *@GuardedBy* به آن اعمال می‌شود، تنها هنگامی می‌تواند مورد دسترسی قرار گیرد که قفل مشخصی را نگه دارد. این قفل می‌تواند یک قفل ذاتی[۱۲۲] یا یک قفل پویا، مانند *java.util.concurrent.Lock*، باشد. به‌عنوان نمونه، کلاس *MovablePoint* زیر، یک نقطه‌ی قابل حرکت را نشان می‌دهد که می‌تواند مکان‌های قبلی خود را با استفاده از لیست آرایه‌ای *memo*، به خاطر بسپارد:

[۱۲۲] Intrinsic

```
@ThreadSafe
public final class MovablePoint
{
	@GuardedBy("this")
	double xPos = 1.0;
	@GuardedBy("this")
	double yPos = 1.0;
	@GuardedBy("itself")
	static final List<MovablePoint> memo=new ArrayList<MovablePoint>();
	public void move(double slope, double distance)
	{
		synchronized (this)
		{
			rememberPoint(this);
			xPos += (1 / slope) * distance;
			yPos += slope * distance;
		}
	}
	public static void rememberPoint(MovablePoint value)
	{
		synchronized (memo)
		{
			memo.add(value);
		}
	}
}
```

حاشیه‌نگاری‌های *@GuardedBy* روی فیلدهای *xPos* و *yPos*، نشان می‌دهند که دسترسی به این فیلدها، با نگهداری یک قفل روی آن، محافظت می‌شود. متد *move()* نیز روی آن، همگام‌سازی انجام می‌دهد و از این‌طریق، این فیلدها را اصلاح می‌نماید. حاشیه‌نگاری *@GuardedBy* روی لیست *memo*، نشان می‌دهد که یک قفل روی شی *ArrayList*، از محتوای آن محافظت می‌کند. متد *rememberPoint()* نیز روی لیست *memo*، همگام‌سازی اجرا می‌نماید.

مساله‌ای در ارتباط با حاشیه‌نگاری *@GuardedBy* وجود دارد. این حاشیه‌نگاری، در نشان دادن اینکه چه زمانی رابطه‌ای بین فیلدهای یک کلاس وجود دارد، با شکست مواجه می‌شود. با استفاده از حاشیه‌نگاری *@RegionLock SureLogic*، که قفل ناحیه‌ی جدیدی برای کلاسی اعلان می‌کند که این حاشیه‌نگاری بر آن اعمال می‌شود، می‌توان بر این محدودیت غلبه کرد. این اعلان، یک قفل با نام جدید می‌سازد که با

یک شی قفل مشخص با ناحیه‌ای از کلاس، مرتبط است. ناحیه می‌تواند تنها زمانی که قفل نگهداری می‌شود، مورد دسترسی قرار گیرد. به‌عنوان نمونه، سیاست قفل‌گذاری *SimpleLock* نشان می‌دهد که همگام‌سازی روی این نمونه، از تمام وضعیت آن محافظت می‌کند:

```
@RegionLock("SimpleLock is this protects Instance")
class Simple { ... }
```

برخلاف *@GuardedBy*، حاشیه‌نگاری *@RegionLock* به برنامه‌نویس اجازه می‌دهد تا نامی صریح و معنادار به سیاست قفل‌گذاری اعطا کند. علاوه بر نام‌گذاری سیاست قفل‌گذاری، حاشیه‌نگاری *@Region*، به یک نام اجازه می‌دهد تا به یک ناحیه از وضعیتی که در حال حفاظت است، داده‌شود. همان‌گونه که در مثال زیر نشان داده شده‌است، نام باعث می‌شود تا تعلق وضعیت و سیاست قفل‌گذاری به یکدیگر، آشکار گردد:

```
@Region("private AircraftPosition")
@RegionLock("StateLock is stateLock protects AircraftPosition")
public final class Aircraft {
	private final Lock stateLock = new ReentrantLock();
	@InRegion("AircraftPosition")
	private long x, y;
	@InRegion("AircraftPosition")
	private long altitude;
	// ...
	public void setPosition(long x, long y) {
		stateLock.lock();
		try
		{
			this.x = x;
			this.y = y;
		}
		finally
		{
			stateLock.unlock();
		}
	}
	// ...
}
```

در این مثال، یک سیاست قفل‌گذاری به نام *StateLock*، برای نشان دادن این‌که قفل‌گذاری روی *stateLock*، از ناحیه‌ی *AircraftPosition* محافظت می‌کند، استفاده شده، که شامل وضعیت قابل تغییر استفاده‌شده برای نمایش محل هواپیما است.

۴,۴,۳ ساخت اشیای قابل تغییر

معمولاً ساخت شی، به‌عنوان استثنا برای سیاست قفل‌گذاری درنظر گرفته می‌شود، زیرا وقتی که اشیا ساخته می‌شوند، ابتدا به‌صورت محدود به نخ هستند. یک شی، به نخی محدود است که از عملگر جدید برای ساخت نمونه‌ی خود استفاده می‌کند. پس از ساخت، شی می‌تواند به‌صورت ایمن، برای سایر نخ‌ها منتشر شود. با این‌وجود، تا زمانی که نخی، که نمونه را ساخته است، به شی اجازه‌ی به اشتراک گذاشته‌شدن ندهد، به اشتراک گذاشته نمی‌شود. روش‌های انتشار ایمن که در *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012] بررسی شده‌اند (TSM01-J. اجازه ندهید این مرجع حین ساخت شی، رها شود)، می‌توانند به‌صورت مختصر، با حاشیه‌نگاری *@Unique(“return”)* بیان شوند. به‌عنوان نمونه، در کد زیر، حاشیه‌نگاری *@Unique(“return”)*، مستند می‌کند که شی برگشتی از سازنده، یک مرجع منحصر به‌فرد است:

```
@RegionLock("Lock is this protects Instance")
public final class Example
{
	private int x = 1;
	private int y;
	@Unique("return")
	public Example(int y)
	{
		this.y = y;
	}
	// ...
}
```

۵,۴,۳ مستندسازی سیاست‌های محدود به نخ

حاشیه‌نگاری‌هایی که توسط Dean Sutherland و William Scherlis ارائه شدند، می‌توانند سیاست‌های محدود به نخ را مستند کنند. روش آنها، امکان وارسی حاشیه‌نگاری‌ها را در برابر کد نوشته‌شده، فراهم می‌کند. به‌عنوان نمونه، حاشیه‌نگاری‌های زیر، هدف طراحی برنامه‌ای را که حداکثر دارای یک نخ ارسال

رخداد AWT[۱۲۳] و چندین نخ محاسباتی است، توصیف می‌کند و نخ‌های محاسباتی اجازه ندارند ساختارهای داده‌ای AWT و یا رخدادها را اداره نمایند:

```
@ThreadRole AWT, Compute
@IncompatibleThreadRoles AWT, Compute
@MaxRoleCount AWT 1
```

۶,۴,۳ مستندسازی پروتکل‌های منتظر اطلاع‌رسانی

یک کلاس وابسته به وضعیت[۱۲۴] باید یا کاملاً پروتکل‌های منتظر اطلاع‌رسانی[۱۲۵] را در معرض زیرکلاس‌ها (و مستندسازی) قرار دهد، یا کلاً زیرکلاس‌ها را از شرکت در آن، منع نماید. طراحی یک کلاس وابسته به وضعیت، حداقل نیاز دارد تا برای وراثت، در معرض صف‌های حالت، قفل‌ها، و مستندسازی پیش‌بینی‌های حالت و سیاست همگام‌سازی قرار گیرد؛ همچنین، ممکن است نیاز باشد تا در معرض متغیرهای وضعیت زیربنایی قرار گیرد (بدترین کاری که یک کلاس وابسته به وضعیت می‌تواند انجام دهد این است که وضعیت خود را برای زیرکلاس‌ها افشا کند، اما پروتکل‌های منتظر اطلاع‌رسانی را مستند نکند؛ این وضعیت، مشابه با کلاسی است که متغیرهای وضعیت خود را افشا می‌کند اما نامتغیرهای خود را مستند نمی‌نماید).

پروتکل‌های منتظر اطلاع‌رسانی باید به اندازه‌ی کافی مستند شوند. هم اکنون، ما در مورد حاشیه‌نگاری‌های مربوط به این زمینه آگاهی نداریم.

۷,۴,۳ کاربرد

حاشیه‌نگاری کد همروند، به مستندسازی و طراحی کمک می‌کند و می‌تواند برای خودکارسازی تشخیص و جلوگیری از وضعیت رقابتی و رقابت داده‌ای، استفاده شود.

---

[۱۲۳] Abstract Window Toolkit

[۱۲۴] State-dependent class

[۱۲۵] Wait-notify protocols

۵,۳ همواره، مقدارحاصل از اجرای متد را به‌عنوان بازخورد، به عقب برگردانید

بهتر است متدها به‌گونه‌ای طراحی شوند که مقداری را به‌عنوان نتیجه برگردانند یا اصطلاحاً، *return* کنند. این مقدار، به توسعه‌دهنده اجازه می‌دهد درباره‌ی وضعیت فعلی شی و یا نتیجه‌ی حاصل از یک عملیات، یادگیری کند. این امر، از سوی سازمان CERT توصیه شده‌است. "EXP00-J. مقادیر بازگشتی از یک متد را نادیده نگیرید"، مقدار بازگشتی آخرین وضعیت اطلاع می دهد.

همچنین، بازخورد می‌تواند از طریق رویه‌های استاندارد یا اشیای استثنا، که از کلاس *Exception* نشات می‌گیرند، ایفای نقش نماید. با این روش، توسعه‌دهنده هنوز می‌تواند اطلاعات دقیقی درباره‌ی نتیجه‌ی متد و اقدامات ضروری برای ادامه، اتخاذ نماید. در این راستا، استثنا باید یک سری اطلاعات را همراه با جزئیات در اختیار داشته‌باشد تا شرایط غیرطبیعی را به‌صورت خلاصه، اطلاع دهد.

واسط‌های کاربری APIها باید از ترکیبی از این روش‌ها (مقادیر بازگشتی متدها، استثنائات) استفاده نمایند تا مشتری‌ها، نتیجه‌ی درست را از اشتباه تشخیص دهند و برای مدیریت دقیق نتایج اشتباه، تشویقی باشد. در مواردی که مقدار خطای پذیرفته‌شده معمول است و قابل به تفسیر نیست، مقدار خطا باید بازگشت داده‌شود. به‌عبارت دیگر، استثنا باید ایجاد شود. یک متد نباید مقداری را که حامل مقدار داده‌ای صحیح و کد خطا همراه است بازگرداند.

در روشی دیگر، یک شی می‌تواند یک متد تست وضعیت را فراهم سازد تا بررسی نماید که آیا در حالت سازگار قرار دارد یا خیر. این دستاورد، تنها در حالت‌هایی مفید خواهد بود که وضعیت شی، قابل تغییر به نوع نخ نیست. این امر، از ایجاد وضعیت رقابتی TOCTOU در بین دو وضعیت احضار اشیای متد تست وضعیت و فراخوانی متدی که به وضعیت اشیا بستگی دارد، جلوگیری به‌عمل می‌آورد. در این فاصله، ممکن است حالت اشیا به‌صورت غیرمنتظره یا مخربانه‌ای، تغییر کند.

متد، مقادیر را بازمی گرداند و/یا کدهای خطا باید به‌دقت، وضعیت اشیا را در یک سطح انتزاع مناسب مشخص نمایند. مشتریان باید بتوانند بر مقادیر ایجادشده تکیه کنند.

۱,۵,۳ نمونه‌کد ناسازگار

در این مثال، اگر متد *updateNode()* بتواند گرهی را در لیست پیوندی پیدا کند، آن را تغییر می‌دهد.

```
public void updateNode(int id, int newValue)
{
	Node current = root;
	while (current != null)
	{
		if (current.getId() == id)
		{
			current.setValue(newValue);
			break;
		}
		current = current.next;
	}
}
```

این متد، برای نشان دادن تغییرات گره، ناموفق عمل می‌کند. در نتیجه، یک فراخواننده نمی‌تواند تعیین نماید که آیا موفق شده‌است یا خیر.

۲,۵,۳ راه‌حل سازگار (بولین)

این کد، در صورتی‌که بتواند گره را تغییر دهد، مقدار *true* و در غیر این‌صورت، مقدار *false* را باز می‌گرداند.

```
public boolean updateNode(int id, int newValue)
{
	Node current = root;
	while (current != null)
	{
		if (current.getId() == id)
		{
			current.setValue(newValue);
			return true; // Node successfully updated
		}
		current = current.next;
	}
	return false;
}
```

۳,۵,۳ راه‌حل سازگار (استثنا)

این کد، هنگامی‌که *Node* تغییریافته را در لیست بیابد، آن را بازمی‌گرداند و در غیر این‌صورت، یک *NodeNotFoundException* ایجاد می‌شود.

```
public Node updateNode(int id, int newValue)
throws NodeNotFoundException {
	Node current = root;
	while (current != null) {
		if (current.getId() == id) {
			current.setValue(newValue);
			return current;
		}
		current = current.next;
	}
	throw new NodeNotFoundException();
}
```

استفاده از استثنائات جهت تعیین شکست می‌تواند یک انتخاب مناسب باشد، اما ایجاد آنها همیشه مناسب نیست. در حالت کلی، یک متد باید یک استثنا را زمانی ایجاد نماید که انتظار دارد موفق شود، اما یک وضعیت غیرقابل برگشت اتفاق می‌افتد یا انتظار می‌رود که یک متد سطح بالاتر در سلسله‌مراتب فراخوانی، شروع به بازیابی نماید.

۴,۵,۳ راه‌حل سازگار (بازگردانی مقدار *Null*)

این راه‌حل، *Node* به‌روزرسانی‌شده را باز می‌گرداند. بنابراین، توسعه‌دهنده می‌تواند به‌سادگی، با بررسی مقدار *null* بازگشت داده‌شده، متوجه شکست عملیات گردد.

```
public Node updateNode(int id, int newValue) {
	Node current = root;
	while (current != null) {
		if (current.getId() == id) {
			current.setValue(newValue);
			return current;
		}
		current = current.next;
	}
	return null;
}
```

۵,۵,۳ کاربرد

فراهم نکردن بازخورد مناسب از طریق ترکیبی از مقادیر بازگشتی، کدهای خطا، و استثنائات، می‌تواند منجر به ایجاد حالتی متناقض و رفتاری غیرمنتظرانه در برنامه شود.

۶,۳ فایل‌ها را با استفاده از ویژگی‌های چندگانه‌ی فایل، شناسایی نمایید

بسیاری از آسیب‌پذیری‌های امنیتی مربوط به فایل‌ها، زمانی ناشی می‌شود که برنامه‌ای بخواهد به شی فایل ناخواسته‌ای دسترسی پیدا کند. این مسئله، اغلب به این دلیل اتفاق می‌افتد که اسامی فایل‌ها، با اشیای زیربنایی فایل ارتباط چندانی ندارند. اسامی فایل‌ها، اطلاعاتی در خصوص ماهیت خود شی ارائه نمی‌دهند. به‌علاوه، هربار که اسم فایل در یک عملیات مورد استفاده قرار می‌گیرد، پیوند میان یک اسم فایل و یک شی فایل، دوباره ارزیابی می‌شود. این ارزیابی مجدد، نوعی شرایط رقابتی با ویژگی زمان بررسی تا زمان استفاده[۱۲۶] را در برنامه کاربردی[۱۲۷] معرفی می‌نماید. اشیا از نوع *java.io.File* و *java.nio.File.path*، تنها زمانی‌که فایل مورد استفاده قرار می‌گیرد، توسط سیستم‌عامل به اشیای زیربنایی فایل متصل می‌شوند.

سازندگان[۱۲۸] فایل و نیز متدهای *java.io.File*، *renameTo()* و *delete()*،تنها بر اسامی فایل جهت تشخیص و شناسایی آن تکیه می‌کنند. این موضوع، برای متدهای *java.nio.File.path.get()*، ایجاد اشیای *path*، و متدهای *move()* و *delete()* مربوط به *java.nio.file.Files()* نیز صادق است. از تمام این متدها با دقت و احتیاط استفاده نمایید.

خوشبختانه، فایل‌ها اغلب می‌توانند علاوه بر اسم فایل، با استفاده از سایر مشخصه‌ها نیز شناخته شوند. برای مثال، با مقایسه‌ی زمان‌های ایجاد و اصلاح یک فایل، اطلاعات مربوط به فایلی که ایجاد و بسته شده‌است، می‌تواند ذخیره گردد و سپس، به‌منظور اعتبارسنجی فایل در صورت نیاز به بازکردن مجدد آن، استفاده شود. مقایسه‌ی مشخصه‌های چندگانه‌ی فایل این احتمال را که فایل دوباره بازشده، همان فایل قبلی باشد، افزایش می‌دهد.

شناسایی فایل برای آن دسته از برنامه‌های کاربردی، که فایل‌های خود را در دایرکتوری‌های امنی نگهداری می‌کنند که توسط صاحب فایل و یا احتمالاً مدیر یک سیستم قابل دسترسی هستند، از اهمیت کمتری برخوردار است.

---

[۱۲۶] TOCTOU
[۱۲۷] application
[۱۲۸] Constructor

۱,۶,۳ نمونه‌کد ناسازگار

در این نمونه‌کد، فایل شناسایی‌شده توسط رشته‌ی *filename* ، باز، پردازش، بسته و سپس، مجدداً برای خواندن باز می‌شود:

```
public void processFile(String filename)
{
	// Identify a file by its path
	Path file1 = Paths.get(filename);
	// Open the file for writing
	try (BufferedWriter bw = new BufferedWriter
	(new OutputStreamWriter(Files.newOutputStream(file1))))
	{
		// Write to file...
	}
	catch (IOException e)
	{
		// Handle error
	}
	// Close the file
	/*
	* A race condition here allows an attacker to switch
	* out the file for another
	*/
	// Reopen the file for reading
	Path file2 = Paths.get(filename);
	try (BufferedReader br = new BufferedReader
	(new InputStreamReader(Files.newInputStream(file2))))
	{
		String line;
		while ((line = br.readLine()) != null)
		{
			System.out.println(line);
		}
	}
	catch (IOException e)
	{
		// Handle error
	}
}
```

به‌دلیل آن‌که پیوند میان اسم فایل و اشیای زیربنایی (پنهان) فایل به هنگام ایجاد *BufferedReader*، مجدداً ارزیابی می‌گردد، این کد نمی‌تواند تضمین کند که فایل بازشده برای خواندن، همان فایل است که

قبلاً برای نوشتن باز شده‌است. ممکن است هکرها یا مهاجمین، فایل امنی را (مثلا با استفاده از یک لینک نمادین) بین فراخوانی اول *close()* و *BufferedReader()* بعد از آن، جایگزین کرده‌باشند.

۲.۶.۳ نمونه‌کد ناسازگار (*File.isSameFile()*)

در این نمونه‌کد، برنامه‌نویس سعی می‌کند با فرخوانی متد *Files.isSameFile()*، اطمینان حاصل نماید که فایل بازشده برای خواندن، همان فایلی است که قبلاً برای نوشتن باز شده‌بود.

```
public void processFile(String filename) {
	// Identify a file by its path
	Path file1 = Paths.get(filename);
	// Open the file for writing
	try (BufferedWriter bw = new BufferedWriter(new
	OutputStreamWriter(Files.newOutputStream(file1)))) {
		// Write to file
	}
	catch (IOException e) {
		// Handle error
	}
	// ...
	// Reopen the file for reading
	Path file2 = Paths.get(filename);
	if (!Files.isSameFile(file1, file2)) {
		// File was tampered with, handle error
	}
	try (BufferedReader br = new BufferedReader(new
	InputStreamReader(Files.newInputStream(file2)))) {
		String line;
		while ((line = br.readLine()) != null) {
		System.out.println(line);
		}
	}
	catch (IOException e) {
		// Handle error
	}
}
```

متأسفانه، Java API فاقد هرگونه تضمینی است که نشان دهد متد *isSameFile()* می‌تواند یکی بودن فایل‌ها را بررسی نماید. اگر هر دو شی *Path* برابر باشند، این متد بدون بررسی وجود یا عدم وجود فایل، مقدار صحیح را باز می‌گرداند. در واقع، ممکن است *isSameFile()* فقط *Path* و مسیرهای مربوط به یکی

بودن دو فایل را بررسی کند و نمی‌تواند در حین دو علیات باز، تشخیص دهد که آیا فایلی در آن مسیر، با فایل دیگری جایگزین شده‌است یا خیر.

۳.۶.۳ راه‌حل سازگار (صفات چندگانه)

این راه‌حل، زمان‌های ایجاد و آخرین تغییرات مربوط به فایل‌ها را جهت افزایش احتمال این‌که فایل بازشده برای خواندن، همان فایلی است که نوشته شده‌بود، بررسی می‌نماید.

```
public void processFile(String filename) throws IOException{
	// Identify a file by its path
	Path file1 = Paths.get(filename);
	BasicFileAttributes attr1 =
		Files.readAttributes(file1, BasicFileAttributes.class);
	FileTime creation1 = attr1.creationTime();
	FileTime modified1 = attr1.lastModifiedTime();
	// Open the file for writing
	try (BufferedWriter bw = new BufferedWriter(new
	OutputStreamWriter(Files.newOutputStream(file1)))) {
		// Write to file...
	}
	catch (IOException e) {
		// Handle error
	}
	// Reopen the file for reading
	Path file2 = Paths.get(filename);
	BasicFileAttributes attr2 =
		Files.readAttributes(file2, BasicFileAttributes.class);
	FileTime creation2 = attr2.creationTime();
	FileTime modified2 = attr2.lastModifiedTime();
	if ( (!creation1.equals(creation2)) ||(!modified1.equals(modified2)) ) {
		// File was tampered with, handle error
	}
	try (BufferedReader br = new BufferedReader(new
	InputStreamReader(Files.newInputStream(file2)))){
		String line;
		while ((line = br.readLine()) != null) {
			System.out.println(line);
		}
	}
	catch (IOException e) {
		// Handle error
	}
}
```

اگرچه که این راه‌حل، معقولانه و امن است، اما یک مهاجم مصمم می‌تواند لینکی نمادین با همان زمان‌های ایجاد و آخرین تغییرات، به‌عنوان فایل اصلی ایجاد کند. همچنین، وضعیت رقابتی TOCTOU می‌تواند بین زمانی که مشخصات فایل برای بار اول خوانده می‌شود و زمانی‌که فایل برای بار اول باز می‌گردد، اتفاق بیفتد. به‌طور مشابه، وضعیت رقابتی TOCTOU دیگری می‌تواند زمانی‌که مشخصات فایل برای بار دوم خوانده می‌شود و فایل مجدداً باز می‌شود، روی دهد.

۴,۶,۳ راه‌حل سازگار (مشخصه‌ی *POSIX fileKey*)

در محیط‌هایی که مشخصه‌ی *fileKey* را پشتیبانی می‌کنند، رویکرد قابل اعتمادتر، بررسی یکی بودن مشخصه‌های *fileKey* مربوط به دو فایل است. مشخصه‌ی *fileKey*، نوعی شی است که به‌طور منحصر به‌فردی، به شناسایی فایل می‌پردازد. این مورد در کد زیر نشان داده شده‌است:

```
public void processFile(String filename) throws IOException{
	// Identify a file by its path
	Path file1 = Paths.get(filename);
	BasicFileAttributes attr1 =
		Files.readAttributes(file1, BasicFileAttributes.class);
	Object key1 = attr1.fileKey();
	// Open the file for writing
	try (BufferedWriter bw = new BufferedWriter(
	new OutputStreamWriter(Files.newOutputStream(file1)))) {
		// Write to file
	}
	catch (IOException e) {
		// Handle error
	}
	// Reopen the file for reading
	Path file2 = Paths.get(filename);
	BasicFileAttributes attr2 =
		Files.readAttributes(file2, BasicFileAttributes.class);
	Object key2 = attr2.fileKey();
	if ( !key1.equals(key2) ) {
		System.out.println("File tampered with");
		// File was tampered with, handle error
	}
```

```
        try (BufferedReader br = new BufferedReader(
        new InputStreamReader(Files.newInputStream(file2)))) {
                String line;
                while ((line = br.readLine()) != null) {
                        System.out.println(line);
                }
        }
        catch (IOException e) {
                // Handle error
        }
}
```

این راه‌حل، برای تمامی پلتفرم‌ها پاسخگو نخواهد بود. برای مثال، تمام صفات *fileKey*، در نسخه‌ی Enterprise ویندوز ۷، تهی هستند.

متد *fileKey()*تضمین می‌کند مقدار کلیدی بازگردانده شده‌ی فایل، تنها زمانی منحصر به‌فرد است که سیستم فایل و فایل‌ها، ایستا باقی بمانند. یک سیستم فایل می‌تواند مجدداً از یک شناسه استفاده نماید: برای مثال، پس از پاک کردن یک فایل. مشابه راه‌حل سازگار پیشین، یک دریچه‌ی رقابتی TOCTOU بین زمانی‌که صفات فایل برای اولین بار خوانده‌می‌شوند و زمانیکه فایل برای اولین بار باز می‌گردد، وجود دارد. وضعیت TOCTOU زمانی اتفاق می‌افتد که صفات برای بار دوم خوانده شده و فایل، مجدداً باز می‌شود.

۵,۶,۳ راه‌حل سازگار (*RandomAccessFile*)

رویکرد بهتر، باز نکردن مجدد فایل‌ها است. این راه‌حل سازگار، نحوه‌ی استفاده از یک کد *RandomAccessFile* را نشان می‌دهد که می‌تواند به دو منظور خواندن و نوشتن، مورد استفاده قرار گیرد. از آنجایی‌که فایل به‌طور خودکار، تنها با دستور *try-with-resources* بسته می‌شود، هیچ وضعیت رقابتی اتفاق نمی‌افتد. توجه داشته‌باشید که این راه‌حل و سایر راه‌حل‌ها، از متد *readLine()* برای مقاصد توضیحی استفاده می‌کنند.

```
public void processFile(String filename) throws IOException
{
	// Identify a file by its path
	try ( RandomAccessFile file = new RandomAccessFile(filename, "rw"))
	{
		// Write to file...
		// Go back to beginning and read contents
		file.seek(0);
		String line;
		while ((line = file.readLine()) != null)
		{
			System.out.println(line);
		}
	}
}
```

۶,۶,۳ نمونه‌کد ناسازگار (اندازه‌ی فایل)

این نمونه‌کد، سعی دارد اطمینان حاصل نماید فایلی که باز می‌شود، دقیقاً حاوی ۱۰۲۴ بایت است.

```
static long goodSize = 1024;
public void doSomethingWithFile(String filename)
{
	long size = new File(filename).length();
	if (size != goodSize)
	{
		System.out.println("File has wrong size!");
		return;
	}
	try (BufferedReader br = new BufferedReader(new InputStreamReader
	(new FileInputStream(filename))))
	{
		// ... Work with file
	}
	catch (IOException e)
	{
		// Handle error
	}
}
```

این کد نیز بین زمانی‌که اندازه فایل بررسی می‌شود و زمانی که فایل باز می‌گردد، در معرض وضعیت رقابتی TOCTOU قرار دارد. اگر مهاجم یا هکر، در طول دریچه‌ی رقابتی، فایل ۱۰۲۴ بایتی را با فایل

دیگری جایگزین کند، می‌تواند با این برنامه، هر فایلی را باز نماید، بررسی را زیر سوال ببرد و به شکست بکشاند.

۷.۶.۳ راه‌حل سازگار (اندازه‌ی فایل)

این راه‌حل، از متد *FileChannel.size()* به‌منظور به‌دست آوردن اندازه‌ی فایل استفاده می کند. از آنجایی که این متد، تنها پس از باز شدن آن فایل به *FileInputStream* اعمال می‌شود، این راه‌حل، دریچه ی رقابتی را از بین می‌برد.

```
static long goodSize = 1024;
public void doSomethingWithFile(String filename)
{
	try (FileInputStream in = new FileInputStream(filename);
	BufferedReader br = new BufferedReader(new InputStreamReader(in)))
	{
		long size = in.getChannel().size();
		if (size != goodSize)
		{
			System.out.println("File has wrong size!");
			return;
		}
		String line;
		while ((line = br.readLine()) != null)
		{
			System.out.println(line);
		}
	}
	catch (IOException e)
	{
		// Handle error
	}
}
```

۸.۶.۳ کاربرد

مهاجمان اغلب برای این‌که از طریق برنامه‌ها به یک فایل ناخواسته دسترسی پیدا کنند، از آسیب‌پذیری‌های مربوط به فایل استفاده می‌نمایند.

۷٫۳ به مقیاس ترتیبی مربوط به یک *enum*، اهمیت چندانی ندهید

انواع شمارش در زبان جاوا، متدی به نام *ordinal()* دارندکه موقعیت عددی هر ثابت شمارشی را در تعریف کلاس آن، باز می‌گرداند.

مطابق با کلاس *Enum<E extends Enum<E>>*، *Public final int ordinal()*، ترتیب ثابت شمارشی را باز می‌گرداند (موقعیت آن در تعریف *enum*، به آنجایی اشاره می‌کند که مقدار اولیه‌ی ثابت صفر به آن اختصاص داده شده‌است). اکثر برنامه‌نویسان، برای این متد، کاربردی نخواهند یافت. این روش، جهت استفاده‌ی ساختارهای داده‌ای *enum* محور پیچیده، همچون *EnumSet* و *EnumMap*، طراحی شده‌است.

مشخصه‌ی زبان جاوا (JLS)[۱۲۹]، استفاده از *ordinal()* را در برنامه‌ها مشخص نمی‌کند. با این‌وجود، اهمیت خارجی دادن به مقدار *ordinal()* مربوط به یک ثابت *enum*، مستعد خطا است و باید در برنامه‌نویسی دفاعی از آن اجتناب نمود.

۱٫۷٫۳ نمونه‌کد ناسازگار

این نمونه‌کد، *enum Hydrocarbon* را تعریف کرده و از متد *ordinal()* مربوط به آن، به‌منظور ارائه‌ی نتیجه‌ی متد *getNumberOfCarbons()*، استفاده می‌کند:

```
enum Hydrocarbon
{
	METHANE, ETHANE, PROPANE, BUTANE, PENTANE,
	HEXANE, HEPTANE, OCTANE, NONANE, DECANE;
	public int getNumberOfCarbons()
	{
		return ordinal() + 1;
	}
}
```

اگرچه که این نمونه‌کد، مطابق انتظار رفتار می‌کند، اما احتمال می‌رود پایداری آن زیر سوال باشد. اگر ثابت‌های *enum*، مجددا مرتب شوند، متد *getNumberOfCarbons()*، مقادیر غیرصحیح را باز می‌گرداند. به‌علاوه احتمالاً، اضافه کردن یک ثابت *BENZENE* اضافه به مدل توسط متد *getNumberOfCarbons()*،

[۱۲۹] Java Language Specification (JLS)

غیرممکن است، زیرا بنزن، شش کربن دارد. درحالی‌که مقدار ترتیبی شش قبلاً برای هگزان درنظر گرفته شده‌است.

۲.۷.۳ راه‌حل سازگار

در این راه‌حل، ثابت‌های *enum*، برای تعداد اتم‌های کربنی که دارند، به‌وضوح به مقادیر صحیح متناظر خود مرتبط شده‌اند.

```
enum Hydrocarbon {
	METHANE(1), ETHANE(2), PROPANE(3), BUTANE(4), PENTANE(5),
	HEXANE(6), BENZENE(6), HEPTANE(7), OCTANE(8), NONANE(9), DECANE(10);
	private final int numberOfCarbons;
	Hydrocarbon(int carbons) { this.numberOfCarbons = carbons; }
	public int getNumberOfCarbons() {
		return numberOfCarbons;
	}
}
```

متد *getNumberOfCarbons()*، از متد *ordinal()* به‌منظور کشف تعداد اتم‌های کربن برای هر مقدار دیگر استفاده نمی‌کند. همچنان‌که در *HEXANE* و *BENZENE* نشان داده شده‌است، ثابت‌های *enum* مختلف می‌توانند به یک مقدار نسبت داده‌شوند. علاوه بر این، این راه‌حل، هیچ‌گونه وابستگی به ترتیب شمارش ندارد، به‌طوری‌که حتی اگر شمارش مجددا مرتب شود، متد *getNumberOfCarbons()*، به‌درستی به کار خود ادامه خواهد داد.

۳.۷.۳ کاربرد

استفاده از ترتیب‌های منتسب به نوع شمارش‌شده، وقتی پذیرفتنی است که ترتیب ثابت‌های شمارشی استاندارد بوده و ثابت‌های اضافی نمی‌توانند اضافه گردند. برای مثال، استفاده از ترتیب‌ها در نوع شمارش‌شده‌ی زیر مجاز است:

```
public enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}
```

به‌طور کلی، استفاده از ترتیب‌ها به‌منظور به‌دست آوردن مقادیر صحیح، پایداری و قابلیت ارتقای برنامه را کاهش می‌دهد و منجر به خطا در برنامه می‌شود.

۸,۳ از رفتار ارتقای عددی آگاه باشید

ارتقای عددی به‌منظور تبدیل عملوندهای یک عملگر به نوع معمول است تا یک عملیات بتواند اجرا شود. به هنگام استفاده از عملگرهای محاسباتی با اندازه‌های عملوند ترکیبی، عملوندهای باریک‌تر، به نوع عریض‌تر آن ارتقا می‌یابند.

۱,۸,۳ قواعد ارتقا

در بخش ارتقای عددی JLS، ارتقای عددی به‌شرح زیر توصیف شده‌است:

- اگر هریک از عملوندها، از یک نوع مرجع برخوردار باشند، تبدیل آنباکس کردن[۱۳۰] اجرا می‌شود.
- اگر یکی از دو عملوند از نوع *double* باشد، دیگری به *double* تبدیل می‌شود.
- در غیر این‌صورت، اگر یکی از دو عملوند از نوع *float* باشد، دیگری به *float* تبدیل می‌شود.
- در غیر این‌صورت، اگر یکی از دو عملوند از نوع *long* باشد، دیگری به *long* تبدیل می‌شود.
- در غیر این‌صورت، اگر یکی از دو عملوند از نوع *int* باشد، دیگری به *int* تبدیل می‌شود.

تبدیل‌های عریض‌کننده‌ی نشأت‌گرفته از ارتقاهای صحیح، اندازه‌ی کلی و نهایی عدد را حفظ می‌کنند. با این‌حال، ارتقاهایی که عملوندها از یک *int* به *float* و یا از *long* به یک *double* تبدیل می‌شوند، می‌توانند باعث ازبین رفتن دقت گردند. این تبدیل‌ها می‌توانند به هنگام استفاده از عملگرهای ضربی (%، *، /)، عملگرهای تجمیعی (+ و -)، عملگرهای مقایسه‌ای (<، <=، >، >=)، عملگرهای بررسی برابری (== و =!) و عملگرهای عددصحیح بیتی (&، |، ^)، روی دهند.

۲,۸,۳ نمونه‌ها

در نمونه‌ی زیر، پیش از آن‌که عملگر + اعمال شود، *a* به یک *double* ارتقا داده‌می‌شود.

```
int a = some_value;
double b = some_other_value;
double c = a + b;
```

[۱۳۰] Unboxing

در نمونه‌ی زیر، ابتدا *b* به *int* تبدیل می‌شود تا عملگر + بتواند به عملوندهای همان نوع اعمال شود.

```
int a = some_value;
char b = some_character;
if ((a + b) > 1.1f)
{
      // Do something
}
```

سپس، نتیجه‌ی *(a+b)* به یک *float* تبدیل می‌شود و در انتها، عملگر مقایسه اعمال می‌گردد.

۳,۸,۳ عملگرهای ترکیبی

ممکن است زمانی‌که عبارات ترکیبی با انواع عملوند ترکیب شده و استفاده می‌شوند، نوع اجباری روی دهد. مثال‌های عملگرهای انتسابی ترکیبی عبارتند از: +=، -=، *=، /=، &=، ^=، %=، <<=، >>=، >>>= و |=.

یک عبارت انتسابی ترکیبی به فرم *E1 op= E2*، برابر است با *E1 = (T) ((E1) op (E2))*، به‌گونه‌ای که *T*، نوع *E1* است؛ با این استثنا که *E1*، تنها یک‌بار ارزیابی می‌شود. این بدان معناست که عبارت انتسابی ترکیبی، به‌طور ضمنی، محاسبه‌ی نهایی را به نوع عملوند سمت چپ، محاسبه می‌کند.

وقتی عملوندها از انواع مختلف هستند، تبدیلات چندگانه روی می‌دهد: برای مثال، زمانی‌که *E1* یک *int* بوده و *E2*، یک *long*، یک *float*، و یا یک *double* است. بنابراین، *E1* نوع *int*، به نوع *E2* (قبل از op) گسترده می‌شود، که با تبدیل باریک‌کننده‌ی نوع *E2* به حالت قبل، یعنی *int*، همراه است (بعد از *op*، اما پیش از انتساب).

۴,۸,۳ نمونه‌کد ناسازگار (ضرب)

در این نمونه، یک متغیر نوع *int* به نام *big*، در یک مقدار نوع *float* به نام *one*، ضرب می‌شود.

```
int big = 1999999999;
float one = 1.0f;
// Binary operation, loses precision because of implicit cast
System.out.println(big * one);
```

در این مورد، پیش از آن‌که ضرب اتفاق بیفتد، *big* به نوع *float* ارتقا داده می‌شود، که این مسئله، از دست رفتن دقت را در پی خواهد داشت. خروجی این کد، *2.0E9* است، در حالی‌که پاسخ صحیح، *1.999999999E9* است.

۵،۸،۳ راه‌حل سازگار (ضرب)

این راه‌حل، از نوع *double* به جای *float* استفاده می‌کند که راهی امن‌تر برای انجام تبدیل عریض‌کننده‌ی ابتدایی است و از ارتقای نوع صحیح ناشی می‌شود.

```
int big = 1999999999;
double one = 1.0d; // Double instead of float
System.out.println(big * one);
```

این راه‌حل، نتیجه‌ی مورد انتظار *1.999999999E9* را به همراه دارد.

۶،۸،۳ نمونه‌کد ناسازگار (شیفت به چپ)

این نمونه‌کد، ارتقای نوع صحیح ناشی از استفاده عملگربیتی *OR* را نشان می‌دهد.

```
byte[] b = new byte[4];
int result = 0;
for (int i = 0; i < 4; i++)
{
	result = (result << 8) | b[i];
}
```

هر عنصر آرایه‌ای بایتی، پیش از آن‌که به‌عنوان عملوند استفاده شود، به بسط علامتی ۳۲ بیتی تبدیل می‌گردد. اگر از ابتدا حاوی مقدار *0xff* باشد، شامل *0xffffffff* نیز می‌شود. این امر باعث می‌شود که *result* شامل مقداری غیر از الحاق ۴ عنصر آرایه شود.

۷،۸،۳ راه‌حل سازگار (شیفت به چپ)

این راه‌حل، ۲۴ بیت بالایی عنصر آرایه ای بایت را می‌کند تا نتیجه‌ی موردنظر را به‌دست آورد.

```
byte[] b = new byte[4];
int result = 0;
for (int i = 0; i < 4; i++)
{
	result = (result << 8) | (b[i] & 0xff);
}
```

۸.۸.۳ نمونه‌کد ناسازگار (انتساب و تجمیع ترکیبی)

این نمونه، یک عملیات انتساب ترکیبی را اجرا می‌کند.

```
int x = 2147483642; // 0x7ffffffa
x += 1.0f; // x contains 2147483647 (0x7fffffff) after the computation
```

عملیات ترکیبی، شامل یک مقدار *int* بوده که حاوی بیت‌های مهم و معنادار بسیاری برای سازگار شدن در عدد اعشاری ۲۳ بیتی یک *float* از جاوا است. این عملیات باعث می‌شود تبدیل از *int* به *float*، منجر به از دست رفتن دقت گردد. غالبا مقدار نهایی، غیرقابل انتظار خواهد بود.

۹.۸.۳ راه‌حل سازگار (انتساب و تجمیع ترکیبی)

برای برنامه‌نویسی‌های دفاعی، از به‌کارگیری هریک از عملگرهای انتسابی ترکیبی روی متغیرهای نوع *byte*، *short* و یا *char*، خودداری کنید. همچنین، از یک عملوند بزرگتر (عریض)، در سمت راست، استفاده نکنید. در این راه‌حل، تمام عملوندها، از نوع *double* جاوا هستند.

```
double x = 2147483642; // 0x7ffffffa
x += 1.0; // x contains 2147483643.0 (0x7ffffffb.0) as expected
```

۱۰.۸.۳ نمونه‌کد ناسازگار (انتساب و شیفت بیتی ترکیبی)

این نمونه، از یک عملگر ترکیبی شیفت راست برای شیفت دادن یک‌بیتی مقدار *i* استفاده می‌کند.

```
short i = -1;
i >>>= 1;
```

متأسفانه، مقدار *i* تغییر نمی‌کند و به‌همان صورت باقی می‌ماند. ابتدا، مقدار *i* به یک *int* ارتقا داده می‌شود. این یک تبدیل عریض‌کننده‌ی ابتدایی است. بنابراین، داده‌ای ازدست نمی‌رود. به‌عنوان یک *short*، ۱- به‌صورت *oxffff* نمایش داده می‌شود. تبدیل به *int*، به مقدار *0xffffffff* منتهی می‌شود و یک بیت به راست شیفت داده می‌شود تا مقدار *0x7fffffff* را نتیجه بدهد. جاوا برای این‌که مقدار را مجددا در متغیر *i*

مربوط به *short* ذخیره نماید، یک تبدیل باریک‌کننده‌ی ضمنی، اجرا می‌کند و بیت‌های مرتبه‌ی بالای ۱۶ را دور می‌اندازد. مجددا، نتیجه‌ی نهایی، *0xffff* یا ۱- خواهد بود.

۱۱.۸.۳ راه‌حل سازگار (انتساب و شیفت بیتی ترکیبی)

این راه‌حل، عملگر انتسابی ترکیبی را به یک *int* اعمال می‌کند که نیاز به عریض و باریک کردن بعدی ندارد. در نتیجه، *i* مقدار *0x7fffffff* را خواهد داشت.

```
int i = -1;
i >>>= 1;
```

۱۲.۸.۳ کاربرد

در نظر نگرفتن ارتقاهای نوع صحیح در مواجهه با عملوندهای این نوع و نیز نوع اعشاری، می‌تواند به از دست رفتن دقت در کار منتهی شود.

۹,۳ بررسی نوع زمان کامپایل انواع پارامترهای متغیر *arity* را فعال کنید.

یک متغیر *arity* (*varargs*)[۱۳۱]، متدی است که می‌تواند تعداد متغیری از آرگومان‌ها را بپذیرد. این متد باید حداقل یک آرگومان ثابت داشته‌باشد. به هنگام پردازش فراخوانی متد متغیر *arity*، کامپایلر جاوا تمام انواع آرگومان‌ها را بررسی می‌کند، به‌طوری‌که تمامی آرگومان‌های واقعی متغیر باید با نوع آرگومان رسمی متغیر، تطبیق داشته‌باشند. با این‌حال بررسی نوع در زمان کامپایل، که *object* یا انواع پارامترهای پویا استفاده می‌شوند، بی اثر و غیرقابل انجام است. حضور پارامترهای ابتدایی مربوط به انواع خاص، اهمیتی نخواهد داشت. کامپایلر همچنان قادر نخواهد بود تا *object* یا انواع پارامتر پویا را بررسی نماید. بررسی قوی نوع در زمان کامپایل متدهای متغیر *arity* را با استفاده از خاص‌ترین نوع ممکن، برای پارامترهای متد، عملی سازید.

۱,۹,۳ نمونه‌کد ناسازگار (شی)

این نمونه، مجموعه‌ای از اعداد را با استفاده از متد متغیر *arity*، که از *object* به‌عنوان متغیر *arity* استفاده می‌کند، با هم جمع می‌نماید. در نتیجه، این متد، ترکیبی دلخواه از پارامترهای هر نوع شی را می‌پذیرد. استفاده‌ی نوع صحیح از چنین تعریفی، بسیار نادرست است.

```
double sum(Object... args) {
	double result = 0.0;
	for (Object arg : args) {
		if (arg instanceof Byte) {
			result += ((Byte) arg).byteValue();
		} else if (arg instanceof Short) {
			result += ((Short) arg).shortValue();
		} else if (arg instanceof Integer) {
			result += ((Integer) arg).intValue();
		} else if (arg instanceof Long) {
			result += ((Long) arg).longValue();
		} else if (arg instanceof Float) {
			result += ((Float) arg).floatValue();
		} else if (arg instanceof Double) {
			result += ((Double) arg).doubleValue();
		} else {
			throw new ClassCastException();
		}
	}
	return result;
}
```

۲.۹.۳ راه‌حل سازگار (عدد)

این راه‌حل، همان متد را تعریف می‌کند، اما از نوع *Number* استفاده می‌نماید. این کلاس انتزاعی، به اندازه‌ی کافی، کلی و عام است تا تمام انواع عددی را در بر گیرد. با این‌حال، به همان اندازه نیز خاص است که انواع غیرعددی را شامل نشود.

```
double sum(Number... args) {// ...}
```

۳.۹.۳ نمونه‌کد ناسازگار (نوع پویا)

این نمونه‌کد، همان متد آرایه متغیر را با استفاده از پارامتری از نوع پویا تعریف می‌کند. این نمونه، تعداد متغیری از پارامترها را، که همه از یک نوع شی هستند، می‌پذیرد. اگرچه، ممکن است هر نوع شی‌ای باشد. استفاده‌ی نوع صحیح از چنین تعاریفی بسیار نادر است.

```
<T> double sum(T... args) {// ...}
```

۴.۹.۳ راه‌حل سازگار (نوع پویا)

این راه‌حل، همان متد پویا را با استفاده از نوع *Number* تعریف می‌کند.

```
<T extends Number> double sum(T... args) {// ...}
```

به هنگام تعریف انواع پارامتر، تا جایی‌که امکان دارد، مشخص و صریح عمل کنید. از به کار بستن *object* و انواع عمومی مبهم در متدهای متغیر *arity* بپرهیزید. همواره، متدهای قدیمی حاوی پارامترهای نهایی آرایه که به متغیر نوع *arity* تبدیل شده‌است، ایده‌ی مناسبی نیست. برای مثال، لغو و متوقف کردن بررسی زمان کامپایل از طریق استفاده از پارامترهای متغیر *arity* در متدی که آرگومان نوع خاصی را نمی‌پذیرد، ممکن خواهد بود تا متد بتواند بدون خطا عمل کامپایل را انجام دهد، که خود باعث به وجود آمدن خطا در زمان‌اجرا خواهد شد.

همچنین، باید توجه داشت که باکسینگ خودکار[۱۳۲]، از بررسی قوی نوع زمان کامپایل انواع ابتدایی و کلاس‌های پنهان‌ساز متناظر، جلوگیری خواهد کرد. برای مثال، این راه‌حل سازگار هشدار ذیل را می‌دهد، اما طبق انتظار عمل می‌کند:

```
Java.java:10: warning: [unchecked] Possible heap pollution from
parameterized vararg type T
<T extends Number> double sum(T... args) {
```

این هشدار کامپایلری خاص می‌تواند بدون ایجاد خطری، نادیده گرفته‌شود.

۵,۹,۳ کاربرد

استفاده‌ی غیرمنطقی از انواع پارامتر متغیر *arity*، از بررسی قوی نوع زمان کامپایل جلوگیری به‌عمل می‌آورد، ابهام ایجاد می‌نماید، و خوانایی کد را تقلیل می‌دهد.

امضاهای متغیر *arity*، با استفاده از *object* و انواع غیردقیق، هنگامی قابل قبول هستند که بدنه‌ی متد، فاقد تبدیلات نوع[۱۳۳] و باکسینگ خودکار باشند و نیز، عمل کامپایل را بدون خطا انجام دهند. مثال زیر را، که به‌درستی بررسی‌های نوع را به‌طور موفقیت‌آمیزی برای تمامی انواع اشیا اعمال می‌کند، درنظر بگیرید:

```
<T> Collection<T> assembleCollection(T... args) {
      return new HashSet<T>(Arrays.asList(args));
}
```

در برخی شرایط، استفاده از یک پارامتر متغیر *arity* نوع *object* ضروری است. یک مثال خوب در این مورد، متد *java.util.Formatter.format(String format, Object... args)* است، که می‌تواند هر نوع اشیا را فرمت نماید. کشف و شناسایی خودکار، بسیار صریح و روشن صورت می‌پذیرد.

---

[۱۳۲] Autoboxing

[۱۳۳] Cast

۱۰٫۳ متد عمومی نهایی را به ثابت‌هایی که ممکن است مقدار آنها در انتشارات بعدی تغییر نماید، اعمال نکنید

کلیدواژه‌ی *final* می‌تواند برای تعیین مقادیر ثابت (مقادیری که نمی‌توانند در طول اجرای برنامه تغییر کنند) استفاده شود. با این‌حال، ثابت‌هایی که می‌توانند در طول عمر یک برنامه تغییر کنند نباید به‌صورت نهایی عمومی تعریف شوند. JLS، اجراها و پیاده‌سازی‌ها را به‌منظور قراردادن مقدار هر فیلد درون‌خطی نهایی عمومی در هر واحد کامپایلی که فیلد را می‌خواند، مجاز می‌شمارد. در نتیجه، اگر کلاس تعریف‌شده، برای این‌که نسخه‌ی جدید مقدار متفاوت را به فیلد بدهد ویرایش شود، واحدهای کامپایلی که فیلد نهایی عمومی را می‌خوانند، تا زمانی‌که دوباره کامپایل شوند، می‌توانند مقدار قدیمی را ببینند. این مشکل ممکن است زمانی اتفاق بیفتد که یک کتابخانه‌ی طرف سوم، به آخرین نسخه، به‌روزرسانی شود، اما کد ارجاع‌دهنده مجدداً کامپایل نشده‌باشد.

یک خطای مرتبط می‌تواند هنگامی رخ دهد که برنامه‌نویس، یک مرجع *static final* را به یک شی ناپایدار تعریف کند.

۱٫۱۰٫۳ نمونه کد ناسازگار

در این نمونه، کلاس *Foo* در *Foo.java*، فیلدی را که نمایان‌گر نسخه‌ی نرم‌افزار است، تعریف می‌کند.

```
class Foo {
	public static final int VERSION = 1;
	// ...
}
```

این فیلد، در مراحل بعد توسط کلاس *Bar* و از طریق یک واحد کامپایلر مجزا (*Bar.java*)، مورد دسترسی قرار می‌گیرد.

```
class Bar
{
	public static void main(String[] args)
	{
		System.out.println("You are using version " + Foo.VERSION);
	}
}
```

پس از کامپایل و اجرا، نرم‌افزار به‌درستی مقدار زیر را چاپ می‌کند:

```
You are using version 1
```

اما اگر توسعه‌دهنده‌ی نرم‌افزار، مقدار *VERSION* را توسط *Foo.java*، به ۲ اصلاح نموده و در نتیجه، *Foo.java* را مجددا کامپایل نماید، در صورتی‌که کامپایل مجدد *Bar.java* با شکست مواجه شود، نرم‌افزار به اشتباه مقدار زیر را چاپ می‌کند:

```
You are using version 1
```

اگرچه، کامپایل مجدد *Bar.java*، این مشکل را حل می‌کند، اما راه‌حل بهتری نیز وجود دارد.

۲.۱۰.۳ راه‌حل سازگار

فارغ از ثابت‌های ریاضی حقیقی، توصیه می‌شود که کد منبع، از آن دسته از متغیرهای کلاس که *static* و *final* را اعلان می‌نمایند، به‌ندرت استفاده کنند. اگر به ماهیت فقط-خواندنی مربوط به *final* نیاز است، گزینه‌ی بهتر، تعریف کردن یک متغیر *private static* و یک متد دست‌یابی مناسب برای به‌دست آوردن مقدار آن است. در این راه‌حل، فیلد *version* در *Foo.java*، به‌صورت *private static* تعریف می‌شود و توسط متد *getversion()*، در دسترس قرار می‌گیرد:

```
class Foo
{
	private static int version = 1;
	public static final int getVersion()
	{
		return version;
	}
	// ...
}
```

کلاس *Bar* در *Bar.java*، تغییر می‌یابد تا متد دست‌یابی *getVersion()* را برای بازیابی فیلد *version* از فایل *Foo.java* فرابخواند:

```
class Bar
{
	public static void main(String[] args)
	{
		System.out.println("You are using version " + Foo.getVersion());
	}
}
```

در این راه‌حل، مقدار نسخه‌ی خصوصی نمی‌تواند وقتی‌که کلاس *Bar* کامپایل می‌شود، در آن کپی گردد. در نتیجه، از اشکال (باگ) جلوگیری به‌عمل می‌اورد. توجه شود که این دگردیسی، هیچ یا به اندازه‌ی کمی، کارایی موردنظر را تحمیل خواهد کرد، زیرا اکثر تولیدکنندگان کدهای (JIT) می‌توانند متد *getVersion()* را در زمان اجرا، روی خط استفاده کنند.

۳.۱۰.۳ کاربرد

تعریف کردن مقداری که در طول عمر نرم‌افزار، به‌عنوان final تغییر می‌کند، می‌تواند منجر به نتایج غیرقابل انتظار شود.

هر تعریف فیلد در بدنه‌ی یک واسط کاربری، به‌طور ضمنی، *static*، *public* و *final* است. تعیین حریم یا تمامی این تغییرات، برای چنین فیلدهایی مجاز است. بنابراین، راهنمایی در مورد فیلدهایی که در واسط‌ها تعریف می‌شوند، صادق نیست. واضح است که اگر مقدار یک فیلد در واسط کاربری تغییر کند، هر کلاسی که از واسط کاربری استفاده می‌کند یا آن را اجرا می‌نماید، باید مجدداً کامپایل شود.

ثابت‌های تعریف‌شده توسط این *enum*، مجازند که از این دستورالعمل تخطی نمایند. ثابت‌هایی که مقدارشان در طول‌عمر این نرم‌افزار هیچ‌گاه تغییر نمی‌کنند، ممکن است به‌عنوان *final* تعریف شوند. برای مثال، JLS توصیه می‌کند که ثابت‌های ریاضی، *final* اعلان شوند.

۱۱,۳ از وابستگی‌های چرخه‌ای بین بسته‌ها اجتناب کنید

ساختار وابستگی یک بسته هیچ‌گاه نباید شامل چرخه باشد. در واقع، باید به‌عنوان گرافی جهت‌دار و غیرمدور (DAG)[۱۳۴]، قابل نمایش باشند. حذف چرخه‌ی بین بسته‌ها، چندین مزیت دارد:

- **آزمایش و قابلیت بهبود و ارتقا**: وابستگی‌های چرخه‌ای، اثرات ناشی از تغییرات یا وصله‌ها[۱۳۵] به کد منبع را بزرگ جلوه می‌دهند، کاهش اثرات تغییرات آزمایش را ساده‌تر می‌سازند، و قابلیت تصمیم و ارتقا را بالا می‌برند. ناتوانی در اجرای آزمایش کافی ناشی از وابستگی‌های چرخه‌ای، دلیل شایع برای آسیب‌پذیری‌های امنیتی است.
- **قابلیت استفاده‌ی مجدد**: وابستگی‌های مدور بین بسته‌ها می‌طلبد که بسته‌ها با نظم و هماهنگی دقیق، انتشار و ارتقا یابند. این شرط، قابلیت استفاده‌ی مجدد را کاهش می‌دهد.
- **شناخت‌ها و انتشارات**: اجتناب از چرخه‌ها کمک می‌کند تا توسعه، به سمت مدولاسیون‌سازی هدایت شود.
- **به کارگیری**: دوری از وابستگی‌های چرخه‌ای بین بسته‌ها، وابستگی[۱۳۶] میان بسته‌ها را کاهش می‌دهد. کاهش وابستگی، بسامد خطاهای زمان‌اجرا، همچون جای خالی، را کاهش می‌دهد. در نتیجه، به کارگیری موثر را ساده‌تر می‌سازد.

۱,۱۱,۳ نمونه کد ناسازگار

این نمونه، شامل بسته‌هایی با نام *account* و *user* است که به‌ترتیب، از کلاس‌های *AccountHolder*، *User*، و *UserDetails* تشکیل شده‌اند. کلاس *UserDetails*، از توسعه‌ی *AccountHolder* ناشی می‌شود، زیرا یک کاربر، نوعی دارنده‌ی حساب است. کلاس *AccountHolder*، به یک متد کاربردی غیرایستای تعریف‌شده در کلاس *User*، وابسته است. به‌طور مشابه، *UserDetails*، به توسعه‌ی *AccountHolder* وابسته است.

---

[۱۳۴] Directed Acyclic Graph (DAG)

[۱۳۵] Patches

[۱۳۶] Coupling

```
package account;
import user.User;
public class AccountHolder {
	private User user;
	public void setUser(User newUser) {user = newUser;}
	synchronized void depositFunds(String username, double amount) {
		// Use a utility method of User to check whether username exists
		if (user.exists(username)) {
			// Deposit the amount
		}
	}
	protected double getBalance(String accountNumber) {
		// Return the account balance
		return 1.0;
	}
}
package user;
import account.AccountHolder;
public class UserDetails extends AccountHolder {
	public synchronized double getUserBalance(String accountNumber) {
		// Use a method of AccountHolder to get the account balance
		return getBalance(accountNumber);
	}
}
public class User {
	public boolean exists(String username) {
		// Check whether user exists
		return true; // Exists
	}
}
```

۲,۱۱,۳ راه‌حل سازگار

همبستگی فشرده‌ی موجود میان کلاس‌های دو بسته، می‌تواند با گنجاندن یک واسط کاربری به نام *BankApplication* در یک بسته‌ی سوم *Bank*، تضعیف شود. وابستگی مدور بسته، با حصول اطمینان از عدم وابستگی *AccountHolder* به *User*، از بین می‌رود، اما در عوض، با وارد نمودن بسته‌ی *Bank*، به واسط کاربری وابسته خواهد بود (نه به اجرای واسط کاربری).

در این راه‌حل، چنین قابلیتی با اضافه کردن یک پارامتر از نوع واسط کاربری (یعنی *BankApplication*) به متد *depositFunds()*، قابل حصول خواهد بود. این راه‌حل، یک قرارداد واقعی را جهت سرمایه‌گذاری،

به *AccountHolder* ارائه می‌دهد. به‌علاوه، *UserDetails*، در حالی‌که سایر متدها را از *AccountHolder* به ارث می‌برد، همزمان، واسط را اجرا می‌نماید و اجراهای واقعی و محسوسی از متدها را ارائه می‌دهد.

```
package bank;
public interface BankApplication {
	void depositFunds(BankApplication ba, String username, double amount);
	double getBalance(String accountNumber);
	double getUserBalance(String accountNumber);
	boolean exists(String username);
}
package account;
import bank.BankApplication; // Import from a third package
class AccountHolder {
	private BankApplication ba;
	public void setBankApplication(BankApplication newBA) {
		ba = newBA;
	}
	public synchronized void depositFunds(BankApplication ba,
	String username, double amount) {
		// Use a utility method of UserDetails to check whether username
		// exists
		if (ba.exists(username)) {
			// Deposit the amount
		}
	}
	public double getBalance(String accountNumber) {
		// Return the account balance
		return 1.0;
	}
}
package user;
import account.AccountHolder; // One-way dependency
import bank.BankApplication; // Import from a third package
public class UserDetails extends AccountHolder implements BankApplication {
	public synchronized double getUserBalance(String accountNumber) {
		// Use a method of AccountHolder to get the account balance
		return getBalance(accountNumber);
	}
	public boolean exists(String username) {
		// Check whether user exists
		return true;
	}
}
```

به‌نظر می‌رسد که واسط *BankApplication*، از مزیت‌های بسیاری، همچون *depositFunds()* و *getBalance()* برخوردار است. این متدها وجود دارند تا اگر زیرکلاسی آنها را نادیده گرفت، آبَرکلاس، قابلیت فراخوانی چندریختی درونی متدهای زیرکلاس را حفظ نماید: به‌عنوان مثال، فراخوانی متد *ba.getBalance()* با یک اجرای نادیده گرفته‌شده از متد در *UserDetails*. نتیجه‌ای از این راه‌حل این است که متدهای تعریف‌شده در واسط، نیاز دارند تا در کلاس‌هایی که آنها را تعریف می‌نمایند، عمومی باشند.

۳.۱۱.۳ کاربرد

وابستگی‌های چرخه‌ای میان بسته‌ها می‌تواند منجر به تولیدات شکننده شود. آسیب‌پذیری امنیتی موجود در یک بسته می‌تواند به‌راحتی به سایر بسته‌ها گسترش یابد.

۱۲,۳ استثنائات تعریف‌شده توسط کاربر را به انواع عام آن ترجیح دهید

از آنجایی‌که یک استثنا با نوع آن ایجاد می‌شود، بهتر است استثنائات را برای مقاصد خاص‌تر و صریح‌تر تعریف نمود تا اینکه از انواع عام آن برای مقاصد چندگانه استفاده کنیم. ایجاد انواع استثنائات عام، درک و ارتقای کد را دشوار می‌سازد و بسیاری از مزایای سازوکار مدیریت استثناءرا از بین خواهد برد.

۱,۱۲,۳ نمونه‌کد ناسازگار

این نمونه‌کد سعی دارد رفتارهای استثنای مختلف را با درنظرگرفتن پیام استثنا، از یک‌دیگر تمییز دهد:

اگر *doSomething()* خطا یا استثنایی را، که نوع آن زیرکلاسی از *Throwable* است، ایجاد نماید، عبارت *switch*، امکان انتخاب یک مورد خاص را برای اجرا فراهم می‌آورد. برای مثال، اگر پیام، "*file not found*" باشد، اقدام مناسب در کد مدیریت-استثنا صورت می‌گیرد.

```
try {
	doSomething();
}
catch (Throwable e) {
	String msg = e.getMessage();
	switch (msg) {
		case "file not found":
			// Handle error
		break;
		case "connection timeout":
			// Handle error
		break;
		case "security violation":
			// Handle error
		break;
		default: throw e;
	}
}
```

با این‌حال، هر تغییری در پیام‌های استثنا منجر به خطا شود، کد را خواهد شکست. برای مثال، فرض کنید کد زیر اجرا گردد:

```
throw new Exception("cannot find file");
```

این استثنا باید توسط جمله‌ی اول، مدیریت شود، اما دوباره ایجاد خواهد شد، زیرا رشته با هیچ جمله‌ای تطبیق ندارد. به‌علاوه، استثنائات می‌توانند بدون یک پیام، ایجاد گردند. استثنای *NullPointerException* فوق و هیچ‌یک از پیشانیان آن را دریافت نکنید، زیرا استثنائات عام را دریافت و مجدداً آنها را ایجاد می‌کنند.

۳,۱۲,۲ راه حل سازگار

این راه‌حل، از انواع استثنای خاص و مشخص استفاده می‌کند و آنها را برای مقاصد خاص جدید، در جاهایی که لازم است، تعریف می‌نماید.

```
public class TimeoutException extends Exception {
	TimeoutException () {
		super();
	}
	TimeoutException (String msg) {
		super(msg);
	}
}
// ...
try {
	doSomething();
}
catch (FileNotFoundException e) {
	// Handle error
}
catch (TimeoutException te) {
	// Handle error
}
catch (SecurityException se) {
	// Handle error
}
```

۳,۱۲,۳ کاربرد

استثنائات، برای مدیریت شرایط استثنایی به کار گرفته می‌شوند. اگر یک استثنا دریافت نشود، برنامه متوقف خواهد شد. هر استثنا که به‌طور ناصحیح یا سطح بازیابی نادرستی دریافت شود، اغلب باعث به‌وجود آمدن رفتار ناصحیح خواهد شد.

۱۳,۳ ۳۴. سعی کنید با دقت و آرامش، از خطاهای سیستمی، بازیابی نمایید

کلاس *RuntimeException* و زیرکلاس‌های آن، و نیز کلاس *Error* و زیر کلاس‌های آن، کلاس‌های استثنای بررسی‌نشده محسوب می‌شوند. تمام کلاس‌های استثنا دیگر، نوع بررسی‌شده هستند.

کلاس‌های استثنایی بررسی‌نشده، در معرض بررسی زمان کامپایل نیستند، زیرا توضیح دادن تمام شرایط استثنایی کاری، زمان‌بر بوده و بازیابی، غالبا دشوار و غیرممکن است. با این‌حال، حتی زمانی‌که بازیابی غیرممکن است، ماشین مجازی جاوا (JVM)، یک خروج بدون دردسر را ممکن می‌سازد و حداقل شانس لازم برای لاگ نمودن خطا را می‌دهد. این‌کار، با استفاده از یک بلاک *try-catch* که *Throwble* را دریافت می‌نماید، صورت می‌گیرد. همچنین، وقتی‌که کد باید به‌طور بالقوه از نشت اطلاعات حساس خودداری کند، دریافت *Throwble* مجاز خواهد بود. در سایر موارد، به‌دلیل آن‌که دریافت *Throwble*، مدیریت استثنائات خاص را دشوار می‌سازد، توصیه نمی‌شود. جاهایی که عملیات پاک‌سازی، همچون آزادسازی منابع سیستم، می‌توانند اجرا شود، خود، باید از یک بلاک *finally* برای آزاد کردن منابع یا یک دستور *try-with-resources* استفاده کند.

۱,۱۳,۳ نمونه‌کد ناسازگار

این نمونه، یک خطای سرریز پشته‌ی *StackOverflowError* است که نتیجه‌ی حاصل از اجرای آن، بازگشت بی‌نهایت است. این‌کار، از تمام فضای پشته‌ی موجود استفاده می‌کند و به انکار خدمات منتهی می‌شود.

```
public class StackOverflow {
	public static void main(String[] args) {
		infiniteRun();
		// ...
	}
	private static void infiniteRun() {
		infiniteRun();
	}
}
```

۲,۱۳,۳ راه‌حل سازگار

این راه‌حل، یک بلاک *try-cath* را، که می‌تواند برای گرفتن خطای *java.lang.Error* یا *java.lang.Throwable* استفاده شود، نشان می‌دهد. در این‌جا، یک ورودی لاگ می‌تواند ایجاد شود، که با تلاش برای آزادسازی منابع کلیدی سیستم در بلاک *finaly* همراه خواهد بود.

```
public class StackOverflow {
	public static void main(String[] args) {
		try {
			infiniteRun();
		}
		catch (Throwable t) {
			// Forward to handler
		}
		finally {
			// Free cache, release resources
		}
		// ...
	}
	private static void infiniteRun() {
		infiniteRun();
	}
}
```

توجه داشته‌باشید که کد *Forward to handler* بایددر وضعیت‌های حافظه محدود، به‌طور صحیح عمل کند، زیرا ممکن است پشته یا هیپ[۱۳۷]، به‌طور کامل استفاده شده‌باشد. در چنین سناریویی، یک تکنیک مفید، رزرو کردن ابتدایی حافظه برای برنامه، به طور خاص به‌منظور استفاده‌ی یک مدیر استثنای بدون حافظه، خواهد بود.

۳,۱۳,۳ کاربرد

اگر به یک خطای سیستمی اجازه داده‌شود تا بتواند برنامه‌ی جاوا را به‌طور ناگهانی متوقف سازد، منجر به آسیب‌پذیری انکار خدمات خواهد شد.

[۱۳۷] Heap

در صورت تمام شدن حافظه، این احتمال می‌رود که بعضی از داده‌های برنامه، در وضعیت ناسازگار قرار بگیرند. در نتیجه، بهترین کار آن است که فرآیند، مجددا راه‌اندازی گردد. اگر تلاش برای ادامه‌ی کار صورت پذیرد، کاهش تعداد نخ‌ها می‌تواند راه حل مفید و کاربردی باشد. این‌کار، در چنین سناریوهایی، کمک‌کننده خواهد بود، زیرا نخ‌ها اغلب، حافظه را نشت داده و با حضور پیوسته‌ی خود، می‌توانند میزان حافظه‌ی برنامه را افزایش دهند.

متدهای *Thread.setUncaughtExceptionHandler()* و *ThreadGroup.uncaughtException()* می‌توانند به‌منظور کمک به مواجهه با یک خطای *OutOfMemoryError* در نخ‌ها، به کارگرفته‌شوند.

۱۴,۳ واسط‌ها را پیش از انتشار، به‌دقت طراحی کنید

واسط‌ها، برای گروه‌بندی تمام متدهای یک کلاس، که قرار است به‌صورت *public* در اختیار عموم قرار گیرد، استفاده می‌شوند. کلاس‌های پیاده‌سازی‌شده باید اجراهای عملی و واقعی را برای تمامی این متدها ارائه دهند. واسط‌ها، قسمت‌های مهمی از اکثر APIهای عمومی هستند. به‌محض انتشار آنها، تعمیر نواقص بدون شکستن هر کدی که نسخه‌ی قدیمی‌تر را اجرا می‌کند، سخت خواهد بود. پیامدهای آن به‌شرح ذیل است:

- تغییرات واسط‌ها از تعمیرات آنها ناشی می‌شود و می‌تواند به‌شدت، قراردادهای کلاس‌های پیاده‌ساز را دچار آسیب نماید. به‌عنوان مثال، ممکن است یک تعمیر اعمال‌شده در نسخه‌ی جدیدتر، با تغییرات و اصلاحات یک واسط غیرمرتبط همراه باشد که باید توسط مشتری اجرا گردد. محتمل است که مشتری، از پیاده‌سازی تعمیر منع شده‌باشد، زیرا واسط جدید، مسئولیت اجرای بیشتری بر وی تحمیل می‌کند.
- پیاده‌سازها می‌توانند پیاده‌سازی پیش‌برد و پایه‌ی مربوط به متدهای واسط را برای مشتریان، ارائه دهند. با این‌حال، چنین کدی می‌تواند اثر سوئی بر رفتار زیرکلاس‌ها داشته‌باشد و بالعکس. وقتی چنین پیاده‌سازی پیش‌فرضی غایب باشد، زیرکلاس‌ها باید پیاده‌سازی‌های بدلی ارائه دهند. این پیاده‌سازی‌ها، محیطی را ترویج می‌دهند که اعلاناتی، همچون "این کد را نادیده بگیر، کاری نمی‌کند"، دائما اتفاق خواهد افتاد. چنین کدی، هیچ‌گاه نمی‌تواند تست شود.
- اگر نفس امنیتی در یکی از نسخه‌های API عمومی وجود داشته‌باشد، در تمام عمر خود API، ادامه خواهد داشت و امنیت هر برنامه‌ی کاربردی یا کتابخانه‌ای را که از آن استفاده می‌کند، تحت تأثیر قرار خواهد داد. حتی پس از آن‌که نقص امنیتی برطرف شود، برنامه‌های کاربردی و کتابخانه‌ها، تا زمانی‌که به‌روز شوند، از نسخه‌ی ناامن استفاده می‌کنند.

۱.۱۴.۳ نمونه‌کد ناسازگار

در این نمونه‌کد، واسط کاربری موسوم به *user*، با دو متد، فریز می‌شود: *Authenticate()* و *subscribe()*. در آینده، ارائه‌دهندگان خدمات، سرویس رایگانی ارائه خواهند داد که به *Authenticate()* وابسته نخواهند بود.

```
public interface User
{
boolean authenticate(String username, char[] password);
void subscribe(int noOfDays);
// Introduced after the class is publicly released
void freeService();
}
```

متأسفانه، اضافه کردن متد *freeservice()*، تمام کدِ مشتری را، که واسط را پیاده‌سازی می‌نماید، خواهد شکست. به‌علاوه، پیاده‌سازهایی که مایلند تنها از *freeservice()* استفاده کنند، باید با مسئولیت ارائه‌ی دو متد دیگر نیز مواجه شوند، که در نتیجه، API را به دلایلی که پیشتر ذکر گردید، آلوده می‌سازند.

۲.۱۴.۳ نمونه‌کد ناسازگار

یک ایده‌ی جایگزین، ترجیح دادن کلاس‌های انتزاعی، برای مواجهه با رشد ثابت آن خواهد بود، اما این‌کار به قیمت نوعی انعطاف‌پذیری است که واسط‌ها ارائه می‌دهند (ممکن است یک کلاس، واسط‌های چندگانه را پیاده‌سازی کند، اما تنها یک کلاس را بسط خواهد داد). توزیع یک کلاس اسکلتی انتزاعی که واسط رو به رشد را پیاده‌سازی می‌نماید، الگوی قابل توجهی برای ارائه‌دهندگان خواهد بود. کلاس اسکلتی می‌تواند به‌صورت گزینشی، تعدادی از متدها را پیاده‌سازی کند و کلاس‌های بسط دهنده را مجبور به ارائه‌ی پیاده‌سازی‌های واقعی از سایر متدها سازد. اگر متد جدیدی به واسط اضافه شود، کلاس اسکلتی می‌تواند پیاده‌سازی پیش‌فرض غیرانتزاعی ارائه دهد. این در حالی است که کلاس بسط‌دهنده می‌تواند به‌صورت اختیاری، لغو کند. این نمونه‌کد، چنین کلاسی را نشان می‌دهد.

با وجود کاربردی بودن، ممکن است این الگو ناامن باشد، زیرا شاید ارائه‌دهنده‌ای که از بسط کد کلاس آگاهی ندارد، آن پیاده‌سازی را انتخاب کند که ضعف‌های امنیتی را در نسخه‌ی مشتری API به‌وجود می‌آورد.

```
public interface User {
	boolean authenticate(String username, char[] password);
	void subscribe(int noOfDays);
	void freeService(); // Introduced after API is
	// publicly released
}
abstract class SkeletalUser implements User {
	public abstract boolean authenticate(String username, char[] password);
	public abstract void subscribe(int noOfDays);
	public void freeService() {
		// Added later, provide implementation and re-release class
	}
}
class Client extends SkeletalUser {
	// Implements authenticate() and subscribe(), not freeService()
}
```

۳.۱۴.۳راه‌حل سازگار (مدوله‌سازی)

یک استراتژی طراحی بهتر، پیش‌بینی آینده‌ی رشد سرویس است. قابلیت اصلی باید در واسط موسوم به *User* پیاده‌سازی شود؛ در این‌صورت، تنها ممکن است به سرویس ابتدایی[۱۳۸] نیاز باشد تا از آن بسط داده‌شود. شاید یک کلاس موجود، تصمیم بگیرد که واسط جدید *FreeUser* را برای استفاده از سرویس رایگان جدید پیاده سازی نماید و یا این‌که آن را کاملا نادیده بگیرد.

```
public interface User {
	boolean authenticate(String username, char[] password);
}
public interface PremiumUser extends User {
	void subscribe(int noOfDays);
}
public interface FreeUser {
	void freeService();
}
```

[۱۳۸] Premium

۴,۱۴,۳راه‌حل سازگار (متد جدید را بلااستفاده سازید)

راه‌حل سازگار، دیگر ایجاد یک استثنا از درون یک متد *freeservice()* جدید تعریف‌شده در زیرکلاس پیاده‌ساز است.

```
class Client implements User {
	public void freeService() {
		throw new AbstractMethodError();
	}
}
```

۵,۱۴,۳راه‌حل سازگار (پیاده‌سازی را به زیرکلاس‌ها بسپارید)

اگرچه مجاز است، اما یک راه‌حل سازگار با انعطاف‌پذیری کم، سپردن اجرای متد، به زیرکلاس‌های کلاس پیاده‌ساز واسط مرکزی متعلق به مشتری است.

```
abstract class Client implements User {
	public abstract void freeService();
	// Delegate implementation of new method to subclasses
	// Other concrete implementations
}
```

۶,۱۴,۳کاربرد

عدم انتشار واسط‌های پایدار و بدون نقص می‌تواند قراردادهای مربوط به کلاس‌های پیاده‌ساز را نقض کند، API مشتری را آلوده سازد، و ضعف‌های امنیتی و کلاس‌های پیاده‌ساز را به‌وجود آورد.

۱۵,۳ کد زباله‌روبی را بنویسید

ویژگی زباله‌روبی (جمع‌آوری داده‌های غیرکاربردی)، از مزیت‌های مهمی برخوردار است. یک زباله‌روب (GC)، به‌منظور بازیابی خودکار حافظه‌ی غیرقابل دسترسی، طراحی شده‌است و از نشت حافظه جلوگیری می‌نماید. اگرچه، GC در اجرای این کار ماهرانه عمل می‌کند، اما با این‌وجود، یک مهاجم حرفه‌ای می‌تواند از طرق مختلف، برای مثال از طریق اختصاص غیرعادی حافظه هیپ یا حفظ شی برای مدت طولانی، یک حمله‌ی انکار خدمات (DoS) علیه GC انجام دهد. برای نمونه، بعضی از نسخه‌های GC، به متوقف کردن تمام نخ‌های اجرایی نیاز دارند تا با درخواست‌های اختصاصی فضا، که منجر به افزایش فعالیت در زمینه‌ی مدیریت هیپ می‌شوند، هماهنگ گردند. در این حالت، خروجی سیستم به‌سرعت کاهش می‌یابد.

سیستم‌های بلادرنگ، نسبت به حمله‌ی دقیق‌تر DoS به نام فرسودگی تدریجی هیپ[۱۳۹]، که از طریق سرقت چرخه‌های CPU صورت می‌گیرد، آسیب پذیرند. یک مهاجم می‌تواند اختصاص حافظه را طوری اجرا نماید که استفاده و مصرف منابع، از قبیل CPU، باتری، حافظه، را بدون به‌وجودآوردن یک خطای *OutOfMemoryError*، افزایش دهد. نوشتن کدهای زباله‌روبی، انجام چنین حملاتی را تا حد بسیاری محدود می‌سازد.

۱,۱۵,۳از اشیای پایدار با عمر کوتاه، استفاده کنید.

مجموعه زباله‌روب‌های نسلی، هزینه‌های جمع‌آوری زباله‌ها را از طریق گروه‌بندی اشیا در نسل‌ها، کاهش می‌دهد. نسخه‌ی جوان‌تر اشیا، با عمر کوتاه تشکیل شده‌است. هنگامی‌که زباله‌روب با اشیای مرده پر می‌شود، یک جمع‌آوری در مقیاس کوچک را روی نسل کوچک اجرا می‌کند. الگوریتم‌های پیشرفته‌ی جمع‌آوری زباله، هزینه‌ی این عمل را به‌قدری کاهش داده‌اند که به‌جای آن‌که متناسب با تعداد اشیای اختصاص داده‌شده از آخرین جمع‌آوری زباله باشند، به تعداد اشیای زنده در نسل جوان‌تر تناسب دارند.

[۱۳۹] Slow-heap-exhostin

شایان ذکر است، اشیای نسل جوان‌تر، که برای زمان‌های طولانی‌تری باقی می‌مانند، تصرف می‌شوند و به نسل تصرف‌شده انتقال می‌یابند. تعدادی از اشیای نسل جوان‌تر، تا دور بعدی جمع‌آوری زباله، به حیات خود ادامه می‌دهند. باقیمانده‌ی آنها نیز برای جمع‌آوری در دوره‌ی آتی آماده می‌گردند.

اساساً، با زباله‌روبی GCهای نسلی، استفاده از اشیای پایدار (نامتغیر) با عمر کوتاه، پربازده‌تر از استفاده از اشیای ناپایدار با عمر بلند است، همانند مخازنی[۱۴۰] شی. اجتناب از مخازن شی، کارایی GC را افزایش می‌دهد. این مخازن، هزینه‌ها و خطرات اضافه‌ای با خود به‌همراه دارند، به‌طوری‌که می‌توانند مشکلات همزمان‌سازی را به‌وجود آورند و نیاز به مدیریت صریح رهاسازی‌ها را پیش آورند، که خود، مشکلاتی را برای اشاره‌گرهای معلق[۱۴۱] به‌وجود خواهد آورد.

به‌علاوه، تعیین مقدار صحیح حافظه برای ذخیره‌ی مخزنی از شی می‌تواند دشوار باشد، مخصوصاً برای کد مأموریت بحرانی[۱۴۲]. استفاده از اشیای ناپایدار با عمر بلند، مخصوصا هنگامی‌که تخصیص اشیا، پرهزینه است، مناسب به‌نظر می‌رسد: به‌عنوان مثال، هنگام اجرای الحاقات چندگانه[۱۴۳] در پایگاه‌های داده. به‌طور مشابه، وقتی اشیا از منابع کمی برخوردار هستند، مخازن شی، به مثابه مخازن نخ و ارتباطات پایگاه داده، انتخاب طراحی مناسبی به‌شمار می‌روند.

۲.۱۵.۳از اشیای بزرگ اجتناب کنید

اختصاص اشیای بزرگ، پرهزینه است. زیرا هزینه‌ی مقداردهی اولیه‌ی فیلدهای آنها، متناسب با اندازه‌ی آنها است. به‌علاوه، اختصاص اشیای بزرگ با اندازه‌های متفاوت می‌تواند مشکلاتی را در تکه‌تکه‌سازی[۱۴۴] یا عملیات‌های فشرده‌سازی جمع‌آوری، به‌وجود آورد.

---

[۱۴۰] Pool
[۱۴۱] Dangling Pointers
[۱۴۲] Mission-critical
[۱۴۳] Multiple joins
[۱۴۴] Fragmentation

۳,۱۵,۳زباله‌روب را به‌صراحت فراخوانی نکنید

زباله‌روب می‌تواند به‌صراحت، با فراخوانی متد *System.gc()* احضار شود. اگرچه، مستندات اعلام می‌دارند که آن، زباله‌روی را اجرا می‌نماید، ضمانتی در خصوص این‌که چه‌وقت GC اجرا می‌شود و یا آیا در واقعیت اجرا می‌گردد یا خیر، وجود ندارد. در حقیقت، فراخوانی، تنها پیشنهاد می‌دهد که GC باید اجرا شود. در اینجا JVM می‌تواند این پیشنهاد را نادیده بگیرد.

استفاده‌ی غیرمسئولانه از این ویژگی می‌تواند به جای آن‌که تا زمان مناسب (زمانی‌که جمع‌آوری زباله، بدون مزاحمت قابل توجه اجرای برنامه، امن است) منتظر بماند، سطح عملکرد سیستم را با اِعمال جمع‌آوری زباله در زمان‌های نامناسب، پائین آورد.

در ماشین مجازی هات‌اسپات جاوا[۱۴۵]، *system.gc()*، یک جمع‌آوری صریح زباله را به اجبار اعمال می‌کند. چنین فراخوانی‌هایی می‌تواند در اعماق کتابخانه‌ها دفن شوند، طوری‌که ردیابی آنها مشکل خواهد بود. برای نادیده گرفتن فراخوانی در چنین مواردی، از پرچم *-XX:+DisableExplicitGC* استفاده کنید. یک چرخه‌ی همزمان کم حجم‌تر می‌تواند برای جلوگیری از توقف‌های طولانی به هنگام اجرای زباله‌روبی کامل، با تعیین پرچم *-XX:ExplicitGCInvokedConcurrent*، فراخوانی شود.

۴,۱۵,۳کاربرد

استفاده‌ی نادرست از کاربردهای زباله‌روبی می‌تواند سطح عملکرد را به‌شدت پائین آورد و در نتیجه، منجر به یک حمله‌ی DoS شود. آسیب‌پذیری‌های Apache Gerenime و Tomocat، که در مارس ۲۰۰۹ گزارش شدند، از اشیای داده‌های کنترل‌کننده[۱۴۶]، موسوم به *PoliclyContext*، ناشی گشتند. این اشیا، در یک نخ قرار داده شده‌بودند و هیچ‌گاه آزاد نگشتند. این امر منجر گردید که اشیای مذکور، برای مدت طولانی‌تر از آن‌چه که لازم بود، در حافظه باقی بمانند.

---

[۱۴۵] Hotspot Virtual Machine (VM)

[۱۴۶] Handeler

وقتی یک برنامه‌ی کاربردی از چندین فاز، از قبیل فاز مقداردهی اولیه و فاز آماده‌سازی، می‌گذرد، ممکن است به فشرده‌سازی هیپ میان فازها احتیاج پیدا کنیم. در چنین مواردی، این احتمال وجود دارد که متد *System.gc()* فراخوانی شود، به‌شرطی‌که دوره‌ی مناسب عاری از رویدادی، بین فازها رخ دهد.

فصل چهارم: قابلیت اطمینان

قابلیت اطمینان، عبارت است از قابلیت یک سیستم یا یک بخش برای اجرای نقش‌های مورد نیاز، تحت شرایط بیان‌شده برای یک دوره‌ی زمانی مشخص. همچنین، می‌توان از پایایی به‌عنوان قابلیت و توانایی محصول نرم‌افزاری برای حفظ سطحی از عملکرد مشخص‌شده هنگام استفاده در شرایط مشخص، یاد کرد.

قابلیت اطمینان نرم‌افزار، فاکتور مهمی است که اطمینان‌پذیری سیستم را تحت تأثیر قرار می‌دهد. این موضوع، با قابلیت اطمینان سخت‌افزاری، که در آن کامل بودن طراحی را به‌جای بی‌نقص بودن در تولید، بازتاب می‌دهد، متفاوت است. خورده شدن و پیر شدن، در نرم‌افزار روی نمی‌دهد. محدودیت‌ها در قابلیت اطمینان، نتیجه‌ی نقص‌ها در شرایط، طراحی، و اجرا است. شکست‌هایی که در نتیجه‌ی این نقص‌ها روی می‌دهند، به روشی که محصول نرم‌افزاری از آن استفاده می‌شود و گزینه‌های از برنامه‌ای که انتخاب می‌گردند، بستگی دارند، نه به زمانی که گذشته است.

در تعریف نخست، قابلیت اطمینان نرم‌افزار به‌عنوان احتمال این‌که نرم‌افزار باعث شکست یک سیستم در یک زمان و تحت شرایط مشخص نخواهد شد، عنوان می‌گردد. احتمالا، نقشی کارکردی است که نه تنها به ورودی‌ها و استفاده‌ی سیستم مربوط می‌شود، بلکه به وجود نقص‌ها در نرم‌افزار نیز مرتبط است. در واقع، ورودی‌های سیستم تعیین می‌کنند که آیا با نقص‌های موجود مواجه می شویم یا خیر. پیچیدگی بالای نرم‌افزار مهمترین عامل در مشکلات اطمینان‌پذیری نرم‌افزاری است.

این دستورالعمل‌ها، متعلق به ویژگی‌های زبان جاوا هستند که به‌راحتی می‌توانند توسط افراد ناآگاه، به اشتباه مورد استفاده قرار گیرند. زبان جاوا، از انعطاف‌پذیری در روش‌هایی که از آن استفاده می‌شود برخوردار است، اما برخی از این استفاده‌ها می‌توانند به کد و تکنیک‌های مبهم، که برای درک و ارتقا دشوار خواهند بود، منجر گردند. برنامه‌نویسان با پیروی از این دستورالعمل‌ها کدی را تولید خواهند کرد که کمتر مستعد اشکالات (باگ‌ها) و شکست‌های زمان اجرا هستند. این فصل، شامل دستور العمل‌هایی می‌شود که:

- به کاهش خطاها کمک می‌کنند و در نتیجه، برای طراحی کدهای قابلیت اطمینان جاوا، مهم و ضروری هستند.
- شامل توصیه‌های خاص مربوط به کد نویسی جاوا، به‌منظور افزایش قابلیت اطمینان نرم‌افزار است.

1.4 شناسه‌ها را در زیرحوزه‌ها، سایه‌دار و مبهم نکنید

استفاده‌ی مجدد از اسامی شناسه‌ها در زیرحوزه‌ها[۱۴۷]، منجر به مبهم‌سازی و سایه‌دار شدن می‌شود. شناسه‌هایی که در حوزه‌ی فعلی مجددا استفاده شده‌اند، می‌توانند آنهایی را که در جاهای دیگر تعریف شده‌اند نیز، غیرقابل دسترسی سازند. در واقع، ابهام نحوی ناشی از مبهم‌سازی یا سایه‌دار کردن، نگهداری و حسابرسی کد منبع را دچار دردسر می‌کند. مخصوصاً هنگامی‌که کد، به دسترسی به هر دو موجودیت نام‌گذاری‌شده‌ی ابتدایی و موجودیت غیرقابل دسترسی، نیاز دارد. این مشکل، زمانی که اسم مجددا استفاده‌شده، در بسته‌ی متفاوت دیگری تعریف می‌گردد، وخامت می‌یابد.

ممکن است یک اسم ساده، در زمینه‌هایی قرار بگیرد که در آن، به‌صورت بالقوه به‌عنوان اسم یک متغیر، یک نوع، و یک بسته تفسیر شود. این موضوع نشان می‌دهد که یک متغیر می‌تواند یک نوع یا یک بسته را مبهم سازد و یک نوع هم می‌تواند، یک اسم بسته را دچار ابهام نماید. از سوی دیگر، سایه‌دار کردن، به یک متغیر اشاره دارد که متغیر دیگری را در یک حوزه، غیرقابل دسترس می‌کند. همچنین، یک نوع می‌تواند نوع دیگر را سایه‌دار نماید.

هیچ شناسه‌ای نباید شناسه دیگر را در حوزه، مبهم یا سایه‌دار سازد. برای مثال، یک متغیر محلی نباید از اسامی فیلد یا متدی از کلاس، نام کلاس، و یا اسم بسته، مجددا استفاده نماید. به‌طور مشابه، کلاس درونی نباید از اسم یک کلاس یا بسته‌ی بیرونی استفاده کند.

لغو کردن و سایه‌دار کردن، هر دو، متفاوت از پنهان سازی هستند، طوری‌که یک عضو قابل دسترسی که باید توسط یک زیرکلاس به ارث برده شده‌باشد، با یک عضو زیرکلاس تعریف‌شده‌ی محلی که همان اسم را اتخاذ کرده‌است اما امضای متد ناسازگار و متفاوتی دارد، جایگزین شود.

---

[۱۴۷] Subscopes

۱.۱.۴ نمونه‌کد ناسازگار(سایه کردن فیلد)

این نمونه، از اسم فیلد نمونه‌ی *val* در حوزه‌ی یک متد نمونه، دوباره استفاده می‌کند.

```
class MyVector
{
	private int val = 1;
	private void doLogic()
	{
		int val;
		//...
	}
}
```

رفتار نهایی می‌تواند به‌عنوان سایه‌دار کردن، طبقه‌بندی شود، طوری‌که متغیر متد، متغیر کلاس را در حوزه‌ی متد، غیرقابل دسترسی می‌کند. برای نمونه، انتساب به *this.val* از درون این متد، بر مقدار متغیر کلاس، موثر نخواهد بود.

۲.۱.۴ راه‌حل سازگار (سایه کردن فیلد)

این راه‌حل، سایه را با تغییر اسم متغیر تعریف‌شده در حوزه‌ی متد، از *val* به *newValue*، از بین می‌برد:

```
class MyVector
{
	private int val = 1;
	private void doLogic()
	{
		int newValue;
		//...
	}
}
```

۳.۱.۴ نمونه‌کد ناسازگار (سایه کردن متغیر)

این نمونه، ناسازگار است، زیرا متغیر *i*، مجددا در محدوده‌ی دومین بلوک حلقه‌ی *for* تعریف شده‌است و بر متغیر *i* تعریف‌شده در کلاس *MyVector*، سایه می‌افکند.

```
class MyVector
{
	private int i = 0;
	private void doLogic()
	{
		for (i = 0; i < 10; i++) {/* ... */}
		for (int i = 0; i < 20; i++) {/* ... */}
	}
}
```

۴,۱,۴ راه‌حل سازگار (سایه کردن متغیر)

در این راه‌حل، شمارنده‌ی *i*، تنها در محدوده‌ی هر بلوک حلقه‌ی *for* تعریف می‌شود.

```
class MyVector
{
	private void doLogic()
	{
		for (int i = 0; i < 10; i++) {/* ... */}
		for (int i = 0; i < 20; i++) {/* ... */}
	}
}
```

۵,۱,۴ کاربرد

استفاده‌ی مجدد از یک اسم در کد، خواندن و ارتقای کار را سخت‌تر می‌ساز، که به ضعف‌های امنیتی منتج می‌شود. یک ابزار خودکار، به‌راحتی می‌تواند استفاده‌ی مجدد از شناسه‌ها را در محدوده‌های کنترل‌شده، شناسایی کند.

۲,۴ در هر تعریف، بیش از یک متغیر تعریف نکنید

تعریف کردن چندین متغیر در یک اعلان، باعث بی‌نظمی در انواع متغیرها و مقادیر ابتدایی آنها می‌شود. به‌طور خاص، هیچ یک از موارد ذیل را در یک اعلان، تعریف ننمایید.

- متغیرها با انواع مختلف
- ترکیب متغیرهای با مقدار اولیه و بدون مقدار اولیه

در مجموع، باید هر متغیر را در خط متعلق به خود متغیر، با توضیحی در مورد نقش آن، تعریف کنید. در حالی‌که نیازی برای هماهنگی با این دستورالعمل وجود ندارد. این‌کار همچنین، در کنوانسیون‌های کدنویسی زبان برنامه‌نویسی جاوا توصیه می‌شود. این دستورالعمل، در موارد زیر صادق است:

- دستورات تعریف متغیر محلی
- اعلانات فیلد
- اعلانات (ثابت) فیلد

۱,۲,۴ نمونه‌کد ناسازگار (مقداردهی اولیه)

این نمونه، ممکن است باعث شود یک برنامه‌نویس یا داور، به اشتباه تصور کند که *i* و *j*، هر دو، به عدد ۱ مقداردهی می‌شوند. این در حالی است که فقط *j* مقداردهی اولیه شده‌است و *i*، بدون مقداردهی اولیه باقی می‌ماند:

```
int i, j = 1;
```

۲,۲,۴ راه‌حل سازگار (مقداردهی اولیه)

در این راه‌حل، واضح است که *i* و *j*، هر دو، به ۱ مقداردهی اولیه می‌شوند.

```
int i = 1; // Purpose of i...
int j = 1; // Purpose of j...
```

۳.۲.۴ راه‌حل سازگار (مقداردهی اولیه)

در این راه‌حل، واضح است که *i* و *j*، هر دو، به ۱ مقداردهی اولیه می‌شوند.

```
int i = 1, j = 1;
```

بدین‌ترتیب، چندین متغیر می‌توانند در یک خط، مقداردهی اولیه شوند، این امر برای متغیرهای موقت کم‌اهمیت، همچون اندیس‌های آرایه‌ای، قابل اجرا است.

۴.۲.۴ نمونه‌کد ناسازگار (انواع مختلف)

در این نمونه، برنامه‌نویس چندین متغیر، از جمله یک آرایه، را در یک خط تعریف می‌کند. تمام نمونه‌های نوع *T*، به متدهای کلاس *Object* دسترسی دارند. با این‌حال، این مسئله به آسانی قابل فراموشی است که وقتی برخی از متدها نادیده گرفته می‌شوند، آرایه‌ها، به برخورد رفتاری خاصی احتیاج دارند.

```
public class Example<T>
{
	private T a, b, c[], d;
	public Example(T in)
	{
		a = in;
		b = in;
		c = (T[]) new Object[10];
		d = in;
	}
}
```

وقتی یکی از متدهای *Object*، به‌عنوان مثال *toString()*، نادیده گرفته‌می‌شود، برنامه‌نویس می‌تواند به‌طور تصادفی، برای نوع *T*، که به‌جای در درنظر گرفتن *c* به‌عنوان ارجاعی به شی نوع *T*، آن را آرایه‌ای از نوع *T* لحاظ نموده‌است، یک پیاده‌سازی ارائه کند.

```
public String toString()
{
	return a.toString() + b.toString() + c.toString() + d.toString();
}
```

با این‌حال، ممکن است قصد برنامه‌نویس این بوده‌باشد که *toString()* را روی هر عنصر از آرایه‌ی *c*، درخواست نماید.

```
// Correct functional implementation
public String toString()
{
	String s = a.toString() + b.toString();
	for (int i = 0; i < c.length; i++)
	{
		s += c[i].toString();
	}
	s += d.toString();
	return s;
}
```

۵,۲,۴ راه‌حل سازگار (نوع مختلف)

این راه‌حل، هر تعریف را در خط آن قرار داده‌است و از نشانه‌گذاری ترجیحی برای تعریف آرایه استفاده می‌کند.

```
public class Example<T>
{
	private T a; // Purpose of a...
	private T b; // Purpose of b...
	private T[] c; // Purpose of c[]...
	private T d; // Purpose of d...
	public Example(T in)
	{
		a = in;
		b = in;
		c = (T[]) new Object[10];
		d = in;
	}
}
```

۶,۲,۴ کاربرد

تعریف متغیرهای چندگانه در هر خط می‌تواند خوانایی کد را کاهش دهد و برنامه‌نویس را دچار سردرگمی کند. وقتی بیش از یک متغیر در یک اعلان تعریف می‌شود، اطمینان حاصل کنید که نوع و مقدار اولیه‌ی هر یک از متغیرها، بدیهی است.

تعاریف اندیس‌های حلقه‌ای باید در یک دستور *for* گنجانده شوند؛ حتی در صورتی‌که این‌کار منجر به تعاریف متغیری شود که فاقد توضیحی در مورد هدف آن متغیر است. نیاز نیست که چنین تعاریفی، در یک خط جدا قرار گیرند. همچنین، توضیح می‌تواند حذف گردد.

```
public class Example
{
	void function()
	{
		int mx = 100; // Some max value
		for (int i = 0; i < mx; ++i )
		{
			/* ... */
		}
	}
}
```

۳,۴ از ثابت‌های نمادین معنادار برای نمایش مقادیر لفظی در برنامه، استفاده کنید

جاوا، استفاده از الفاظ مختلف، مانند اعداد صحیح (۲، ۵)، اعداد اعشاری (۲/۵، ۶/۰۲۲e+۲۳)، کاراکترها (‘a’، ‘\n’)، بولین‌ها (true, false) رشته‌ها ("Hello\n")، را پشتیبانی می‌کند. استفاده‌ی گسترده از الفاظ در یک برنامه می‌تواند به دو مشکل منجر گردد: نخست، معنی لفظ، اغلب در محتوا مبهم و ناواضح است؛ دوم، تغییر یک لفظ بسیار استفاده‌شده، خواستار جست‌وجوی کامل منبع برنامه برای آن لفظ و نیز تمییز دادن استفاده‌هایی که باید تغییر داده‌شوند از آنهایی است که باید بدون تغییر باقی بمانند.

با تغییر متغیرهای کلاس با ثابت‌های نام‌گذاری‌شده‌ی معنادار، با قرار دادن مقایر آنها به الفاظ مورد انتظار، و ارجاع ثابت‌ها به جای الفاظ در برنامه، از این مشکلات جلوگیری کنید. این رویکرد به‌روشنی، معنی و استفاده‌ی موردنظر از هر لفظ را نشان می‌دهد. به‌علاوه، اگر ثابت به اصلاح نیاز داشته‌باشد. تغییر باید به تعریف، محدود شود، طوری‌که جست‌وجوی کد، غیرضروری باشد.

ثابت‌ها باید به عنوان *static* و *final* تعریف شوند. با این‌حال، اگر مقادیر آنها تغییر یابد، ثابت‌ها نباید به‌صورت *public* و *final* تعریف شوند: برای مثال

```
private static final int SIZE = 25;
```

اگرچه، *final* می‌تواند برای تعیین ثابت‌های پایدار استفاده گردد، اما باید به هنگام مواجه با اشیای مرکب، با احتیاط عمل شود.

۱,۳,۴ نمونه‌کد ناسازگار

این نمونه، ابعاد تقریبی یک کره را، با درنظر گرفتن شعاع آن، محاسبه می‌کند.

```
double area(double radius) {
	return 3.14 * radius * radius;
}
double volume(double radius) {
	return 4.19 * radius * radius * radius;
}
double greatCircleCircumference(double radius) {
	return 6.28 * radius;
}
```

متدها، از الفاظ به‌ظاهر دلخواه ۳/۱۴، ۴/۱۹، و ۶/۲۸ برای نمایش فاکتورهای مقیاس‌گذاری مختلف استفاده‌شده، برای محاسبه‌ی این ابعاد استفاده می‌کنند. یک طراح یا پشتیبان که این کد را می‌خواند، ممکن است چیز زیادی در مورد این که چطور اینها به‌وجود آمدند و چه معنایی می‌دهند، نداند. در نتیجه، کارکرد این کدها را درک نخواهد کرد.

۲،۳،۴ نمونه‌کد ناسازگار

این نمونه‌ی معنی‌دار، با محاسبه‌ی صریح ثابت‌های مورد نیاز، از این مشکل جلوگیری می‌کند.

```
double area(double radius) {
	return 3.14 * radius * radius;
}
double volume(double radius) {
	return 4.0 / 3.0 * 3.14 * radius * radius * radius;
}
double greatCircleCircumference(double radius) {
	return 2 * 3.14 * radius;
}
```

این کد، برای نمایش مقدار π از لفظ ۳/۱۴ استفاده می‌نماید. اگرچه، این کار مقداری از ابهام لفظ را برطرف می‌سازد، اما ارتقا و بهبود کد را دشوار و پیچیده خواهد کرد. اگر برنامه‌نویس تصمیم بگیرد که مقدار دقیق‌تری را به π اختصاص دهد، تمام تکرارهای ۳/۱۴ در این کد باید پیدا و جایگزین می‌شدند.

۳،۳،۴ راه‌حل سازگار (ثابت‌ها)

در این راه‌حل، به یک ثابت *PI* تعریف‌شده، مقدار اولیه‌ی ۳/۱۴ تخصیص داده می‌شود. پس از این، هر وقت به مقدار π نیاز باشد، به آن ارجاع داده می‌شود.

```
private static final double PI = 3.14;
double area(double radius) {
	return PI * radius * radius;
}
double volume(double radius) {
	return 4.0/3.0 * PI * radius * radius * radius;
}
double greatCircleCircumference(double radius) {
	return 2 * PI * radius;
}
```

این تکنیک، بی‌نظمی را کاهش می‌دهد و قابلیت ارتقا و تعمیر را بالا می‌برد. اگر به یک عدد تقریبی دقیق‌تر از π نیاز باشد، برنامه‌نویس می‌تواند آن ثابت را مجددا تعریف کند. استفاده از الفاظ ۴/۰، ۳/۰، و ۲، از این دستورالعمل تخطی نمی‌کند.

۴.۳.۴ راه‌حل سازگار (ثابت‌های از پیش تعریف‌شده)

در صورت دسترسی و وجود، از ثابت‌های از پیش تعریف‌شده استفاده کنید. کلاس *java.lang.Math* گروه بزرگی از ثابت‌های عددی، از جمله *PI* و ثابت نمایی *E*، را تعریف می‌نماید.

```
double area(double radius) {
	return Math.PI * radius * radius;
}
double volume(double radius) {
	return 4.0/3.0 * Math.PI * radius * radius * radius;
}
double greatCircleCircumference(double radius) {
	return 2 * Math.PI * radius;
}
```

۵.۳.۴ نمونه‌کد ناسازگار

این نمونه، یک ثابت *BUFSIZE* را تعریف می‌کند، اما سپس، هدف از تعریف *BUFSIZE* به‌عنوان یک ثابت را با اتخاذ یک مقدار مشخص برای *BUFSIZE* در عبارت پائین، زیر سوال می‌برد.

```
private static final int BUFSIZE = 512;
// ...
public void shiftBlock()
{
	int nblocks = 1 + ((nbytes - 1) >> 9); // BUFSIZE = 512 = 2^9
	// ...
}
```

برنامه‌نویس فرض کرده که مقدار *BUFSIZE*، ۵۱۲ است و ۹ بیت شیفت به راست (برای اعداد مثبت) همان تقسیم بر ۵۱۲ است. با این‌حال، اگر *BUFSIZE* در آینده به ۱۰۲۴ تغییر یابد، اصلاحات مشکل و مستعد خطا خواهند بود.

جایگزین کردن یک عملیات تقسیم با یک شیفت به راست، یک بهینه‌سازی زودهنگام درنظر گرفته می‌شود. معمولاً وقتی که این باید بهینه‌سازی انجام گیرد، کامپایلر کار تعیین را بهتر انجام می‌دهد.

۶،۳،۴ راه‌حل سازگار

این راه‌حل، از شناسه انتساب داده‌شده به مقدار ثابت در این عبارت استفاده می‌کند.

```
private static final int BUFSIZE = 512;
// ...
public void shiftBlock(int nbytes)
{
      int nblocks = 1 + (nbytes - 1) / BUFSIZE;
      // ...
}
```

۷،۳،۴ کاربرد

استفاده از الفاظ عددی، کد را برای خواندن، فهمیدن، و ویرایش نمودن، سخت‌تر می‌سازد.

استفاده از ثابت‌های نمادین باید به مواردی که خوانایی و قابلیت ارتقای کد را بهبود می‌بخشند، محدود شود. وقتی قصد لفظ واضح است، یا جایی‌که احتمال تغییر لفظ وجود ندارد، استفاده از ثابت‌های نمادین، خوانایی کد را دچا مشکل می‌کند. کد زیر، معنای کد را با استفاده از ثابت‌های نمادین، مبهم کرده‌است.

```
private static final double FOUR = 4.0;
private static final double THREE = 3.0;
double volume(double radius)
{
      return FOUR / THREE * Math.PI * radius * radius * radius;
}
```

مقادیر ۴/۰ و ۳/۰ در محاسبه‌ی دوم، به‌وضوح، فاکتورهای مقیاس‌گذاری استفاده‌شده برای محاسبه‌ی حجم کره هستند و در معرض تغییر قرار ندارند (برخلاف مقدار تقریبی π). بنابراین، می‌توانند به‌طور دقیق

نمایش داده‌شوند. دلیلی برای تغییر آنها جهت افزایش دقت، وجود ندارد، زیرا جایگزین کردن آنها با ثابت‌های نمادین، خوانایی کد را دچار آسیب می‌کند.

۴,۴ روابط را در تعاریف ثابت، به‌درستی کدگذاری کنید

تعاریف عبارات ثابت باید دقیقا وقتی مقادیر بیان‌شده توسط آنها نیز مرتبط است، مرتبط باشند.

۱,۴,۴ نمونه‌کد ناسازگار

در این نمونه، *OUT_STR_LEN* باید همیشه دقیقا ۲ واحد بزرگتر از *IN_STR_LEN* باشد. این تعاریف نمی‌توانند بازتاب‌دهنده‌ی این شرایط باشند.

```
public static final int IN_STR_LEN = 18;
public static final int OUT_STR_LEN = 20;
```

۲,۴,۴ راه‌حل سازگار

در این راه‌حل، رابطه‌ی میان دو مقدار، در تعاریف نشان داده می‌شود.

```
public static final int IN_STR_LEN = 18;
public static final int OUT_STR_LEN = IN_STR_LEN + 2;
```

۳,۴,۴ نمونه‌کد ناسازگار

در این نمونه، به‌نظر رابطه‌ای زیربنایی بین دو ثابتی که هیچ‌یک وجود خارجی ندارند، دیده می‌شود.

```
public static final int VOTING_AGE = 18;
public static final int ALCOHOL_AGE = VOTING_AGE + 3;
```

ممکن است برنامه‌نویسی که ارتقای تعمیر روتین را انجام می‌دهد، تعریف *VOTING_AGE* را اصلاح نماید، اما موفق به تشخیص نتیجه‌ی حاصل از آن در تعریف *ALCOHOL_AGE* نشود.

۴.۴.۴ راه‌حل سازگار

در این راه‌حل، تعاریف، بازتاب‌دهنده‌ی استقلال دو ثابت هستند.

```
public static final int VOTING_AGE = 18;
public static final int ALCOHOL_AGE = 21;
```

۵.۴ یک آرایه یا مجموعه‌ی خالی را به جای یک مقدار *null*، برای متدهایی که یک آرایه یا مجموعه را بازمی‌گردانید، بازگردانید

بعضی از APIها، برای آن‌که نشان دهند نمونه‌ها غیرقابل دسترس هستند، عمداً یک مرجع *null* را بازمی‌گردانند. این‌کار، وقتی‌که کد مشتری موفق به مدیریت صریح مقدار بازگردانی *null* نمی شود، منجر به آسیب‌پذیرهای DoS می‌گردد. یک مقدار بازگردانی *null*، نمونه‌ای از یک شاخص خطای در باند است. برای متدهایی که مجموعه‌ای از مقادیر را توسط یک آرایه یا مجموعه بازمی‌گردانند، بازگردانی یک آرایه یا مجموعه‌ی خالی، جایگزین بسیار خوبی برای بازگردانی یک مقدار *null* است، زیرا اکثر فراخواننده‌ها، برای کنترل مجموعه‌ی خالی، مجهزتر هستند تا یک مقدار *null*.

۱.۵.۴ نمونه‌کد ناسازگار

این نمونه، یک ArrayList را که *null* است، در زمانی‌که اندازه‌ی *Array-List* صفر است، بازمی‌گرداند. کلاس *Inventory*، شامل یک متد *getStock()* است که لیستی از آیتم‌های حاوی *inventory* صفر و نیز لیست آیتم‌ها را به فراخواننده، بازمی‌گردانند.

```
class Inventory {
	private final Hashtable<String, Integer> items;
	public Inventory() {
		items = new Hashtable<String, Integer>();
	}
	public List<String> getStock() {
		List<String> stock = new ArrayList<String>();
		Enumeration itemKeys = items.keys();
		while (itemKeys.hasMoreElements()) {
			Object value = itemKeys.nextElement();
			if ((items.get(value)) == 0) {
				stock.add((String)value);
			}
		}
		if (items.size() == 0) {
			return null;
		}
		else {
			return stock;
		}
	}
}
```

```
public class Client {
	public static void main(String[] args) {
		Inventory inv = new Inventory();
		List<String> items = inv.getStock();
		System.out.println(items.size());
	}
}
```

وقتی اندازه‌ی این لیست، صفر است، با فرض این‌که مشتری، بررسی‌های لازم را اجرا خواهد کرد، مقدار *null* بازگردانده می‌شود. در این نمونه، *NullPointerException* فاقد هر نوع بررسی مقدار *null* است.

۲،۵،۴ راه‌حل سازگار

این راه‌حل، به جای بازگردانی یک مقدار *null*، تنها *List* را باز می‌گرداند، حتی زمانی‌که خالی است.

```
class Inventory {
	private final Hashtable<String, Integer> items;
	public Inventory() {
		items = new Hashtable<String, Integer>();
	}
	public List<String> getStock() {
		List<String> stock = new ArrayList<String>();
		Integer noOfItems; // Number of items left in the inventory
		Enumeration itemKeys = items.keys();
		while (itemKeys.hasMoreElements()) {
			Object value = itemKeys.nextElement();
			if ((noOfItems = items.get(value)) == 0) {
				stock.add((String)value);
			}
		}
		return stock; // Return list (possibly zero-length)
	}
}
public class Client {
	public static void main(String[] args) {
		Inventory inv = new Inventory();
		List<String> items = inv.getStock();
		System.out.println(items.size());
	}
}
```

مشتری می‌تواند این وضعیت را به‌نحو موثری، بدون آن‌که توسط استثنائات دچار اختلال شود، مدیریت نماید. به هنگام بازگردانی آرایه‌ها به جای مجموعه‌ها، مطمئن شوید که مشتری از تلاش برای دسترسی به عناصر مجزای یک آرایه با طول صفر، خودداری کند. این‌کار، از ایجاد *ArrayIndexOutOfBoundsException* جلوگیری به‌عمل می‌آورد.

۳.۵.۴ راه‌حل سازگار

این راه‌حل، یک لیست خالی را به‌سرعت بازمی‌گرداند، که یک تکنیک معادل مجاز به‌شمار می‌رود.

```
public List<String> getStock() {
	List<String> stock = new ArrayList<String>();
	Integer noOfItems; // Number of items left in the inventory
	Enumeration itemKeys = items.keys();
	while (itemKeys.hasMoreElements()) {
		Object value = itemKeys.nextElement();
		if ((noOfItems = items.get(value)) == 0) {
			stock.add((String)value);
		}
	}
	if (l.isEmpty()) {
		return Collections.EMPTY_LIST; // Always zero-length
	}
	else {
		return stock; // Return list
	}
}
// Class Client ...
```

۴.۵.۴ کاربرد

ممکن است بازگردانی یک مقدار *null* به جای یک آرایه با طول صفر یا مجموعه، زمانی‌که کد مشتری موفق به مدیریت صحیح مقادیر بازگردانی *null* نشود، آسیب‌پذیری کدهای DoS را به‌همراه داشته‌باشد.

یک شناسایی و کشف خودکار، روشی آسان است، زیرا عموماً حل مشکل، به دخالت خود برنامه‌نویس نیاز ندارد.

۶,۴ از استثنائات فقط برای شرایط استثنایی استفاده کنید

استثنائات، تنها باید برای نشان دادن وضعیت‌های استثنایی به کار روند و نباید برای مقایسه‌ی جریان کنترلی عادی استفاده گردند. احتمال دارد دریافت شی عمومی همچون *Throwable*، خطاهای غیرمنتظره را دریافت نماید. زمانی‌که برنامه‌ای نوع خاصی از استثنا را دریافت می‌کند، همیشه از نقطه‌ای که آن استثنا ایجاد شده‌است، آگاهی ندارد. استفاده از یک عبارت *cathc* برای کنترل استثنایی که در یک جای شناخته‌شده و دور از دسترس اتفاق می‌افتد، راه حل نامناسبی است. بهتر است که خطا به‌محض وقوع، مدیریت و یا در صورت امکان، از آن جلوگیری شود.

غیرقانونی بودن دستورات *throw* مربوط به دستورات *catch* متناظر می‌تواند مانع از بهینه‌سازی، در راستای بالا بردن سطح کد وابسته به کنترل استثنا، گردد. وابستگی جریان کنترلی به دریافت استثنائات، فرایند اشکال‌زدایی را پیچیده‌تر می‌سازد، زیرا استثنائات، پرشی از دستور *throw* تا *catch* را در جریان کنترلی نشان می‌دهند. در نهایت، لازم نیست استثنائات در سطح بالایی بهینه شوند، زیرا فرض بر آن است که آنها، تنها در شرایط استثنایی ایجاد می‌شوند. *throw* و *catch* کردن یک استثنا، اغلب از عملکرد ضعیف‌تری نسبت به کنترل خطا از طریق سایر مکانیزم‌ها برخوردار هستند.

۱,۶,۴ نمونه‌کد ناسازگار

این نمونه‌کد تلاش دارد تا عناصر پردازش‌شده آرایه‌ی *strings* را الحاق کند. این کد، از یک *ArrayIndexOutOfBoundsException* برای برای شناسایی پایان یک آرایه استفاده می‌کند. متأسفانه، از آنجایی‌که *ArrayIndexOutOfBoundsException* یک *RuntimeException* است، می‌تواند بدون اعلان

در یک عبارت *throws*، توسط متد *processSingleString()* ایجاد شود. بنابراین، متد *processStrings()* قادر خواهد بود به یک‌باره، پیش از پردازش تمام رشته‌ها، خاتمه یابد.

```
public String processSingleString(String string) {
	// ...
	return string;
}
public String processStrings(String[] strings) {
	String result = "";
	int i = 0;
	try {
		while (true) {
			result = result.concat(processSingleString(strings[i]));
			i++;
		}
	}
	catch (ArrayIndexOutOfBoundsException e) {
		// Ignore, we're done
	}
	return result;
}
```

۲.۶.۴ راه‌حل سازگار

این راه‌حل، از یک استاندارد برای حلقه‌ی *for*، به‌منظور الحاق رشته‌ها استفاده می‌کند.

```
public String processStrings(String[] strings) {
	String result = "";
	for (int i = 0; i < strings.length; i++) {
		result = result.concat(processSingleString(strings[i]));
	}
	return result;
}
```

این کد، نیازی به دریافت *ArrayIndexOutOfBoundsException* ندارد، زیرا یک استثنا در زمان اجرا است و چنین استثنائاتی، خطاهای برنامه‌نویسی را نشان می‌دهند که با اصلاح نقص، به بهترین نحو ممکن حل می‌شوند.

۳.۶.۴ کاربرد

استفاده از استثنا برای هر قصدی غیر از کشف و کنترل شرایط استثنایی، تحلیل برنامه و اشکال‌زدایی را پیچیده می‌سازد، عملکرد را کاهش می‌دهد، و هزینه‌ی اصلاح و ارتقا را بالا می‌برد.

۷.۴ از دستور *try-with-resources* برای مدیریت امن منابع بسته‌شدنی استفاده کنید

نسخه‌ی 1.7 از JDK، دستور *try-with-resources* را، که استفاده‌ی درست از منابع موجود در واسط موسوم به *java.lang.AutoCloseable* (شامل آنهایی که واسط *java.io.Closeable* را پیاده‌سازی می‌کنند) ساده می‌سازد، معرفی نمود.

استفاده از این دستور، از مشکلاتی که می‌توانند با یک بلاک معمولی *try-catch-finally*، هنگام بستن منابع پیش آیند جلوگیری می‌نماید. مشکلاتی همچون عدم توانایی در بستن یک منبع، زیرا یک استثنا، در نتیجه‌ی بستن یک منبع دیگر، ایجاد می‌شود؛ یا پوشاندن یک استثنای مهم هنگام بسته شدن یک منبع. فایل‌های موقت را پیش از پایان، حذف کنید و منابع را، وقتی‌که دیگر به آنها نیازی نیست، ببندید.

۱.۷.۴ نمونه‌کد ناسازگار

این نمونه، از یک بلاک *try-catch-finally* معمولی برای بستن دو منبع استفاده می‌کند.

```
public void processFile(String inPath, String outPath)
throws IOException{
	BufferedReader br = null;
	BufferedWriter bw = null;
	try {
		br = new BufferedReader(new FileReader(inPath));
		bw = new BufferedWriter(new FileWriter(outPath));
		// Process the input and produce the output
	}
	finally {
		try {
			if (br != null) {
				br.close();
			}
			if (bw != null) {
				bw.close();
			}
		}
		catch (IOException x) {
			// Handle error
		}
	}
}
```

با این‌وجود، اگر یک استثنا ایجاد شود، وقتی‌که *BufferedReader br* بسته می‌شود، *BufferedWriter bw* بسته نخواهد شد.

۲.۷.۴ راه‌حل سازگار (بلاک *finally* دوم)

این راه‌حل، از یک بلاک *finally* دوم برای تضمین بسته‌شدن درست *bw*، حتی وقتی یک استثنا در زمان بستن *br* ایجاد می‌شود، استفاده می‌نماید.

```
public void processFile(String inPath, String outPath)
throws IOException {
	BufferedReader br = null;
	BufferedWriter bw = null;
	try {
		br = new BufferedReader(new FileReader(inPath));
		bw = new BufferedWriter(new FileWriter(outPath));
		// Process the input and produce the output
	}
	finally {
		if (br != null) {
			try {
				br.close();
			}
			catch (IOException x) {
				// Handle error
			}
			finally {
				if (bw != null) {
					try {
						bw.close();
					}
					catch (IOException x) {
						// Handle error
					}
				}
			}
		}
	}
}
```

۳.۷.۴ راه‌حل سازگار (*try-with-resources*)

این راه‌حل، از یک دستور *try-with-resources* به‌منظور مدیریت *br* و نیز *bw*، استفاده می‌کند.

```
public void processFile(String inPath, String outPath)
throws IOException{
	try (BufferedReader br = new BufferedReader(new FileReader(inPath));
	BufferedWriter bw = new BufferedWriter(new FileWriter(outPath));) {
		// Process the input and produce the output
	}
	catch (IOException ex) {
		// Print out all exceptions, including suppressed ones
		System.err.println("thrown exception: " + ex.toString());
		Throwable[] suppressed = ex.getSuppressed();
		for (int i = 0; i < suppressed.length; i++) {
			System.err.println("suppressed exception: " +
				suppressed[i].toString());
		}
	}
}
```

این راه‌حل، استثنائاتی را که در طول پردازش ورودی ایجاد می‌شوند، حفظ می کند. این در حالی است که همچنان تضمین می‌نماید که هر دوی *br* و *bw*، بدون درنظر گرفتن استثنائاتی که روی می‌دهند، به‌درستی بسته می‌شوند. در نهایت، این کد چگونگی دسترسی به هر استثنایی را که ممکن است از بلاک *try-with-resources* به‌وجود آید، نشان می‌دهد.

اگر تنها یک استثنا در طول بازکردن، پردازش، و بستن ایجاد گردد، استثنای دوم در حالی‌که تلاش می‌کند هر دو فایل را ببندد، ایجاد می‌شود. استثنای دوم و اول، به‌ترتیب، پس از "*thrown exception:*" و "*suppressed exception:*" چاپ خواهندشد.

۴.۷.۴ کاربرد

ممکن است عدم کنترل صحیح تمام موارد شکست در هنگام کار کردن با منابع بسته‌شدنی، به این منتج گردد که برخی از منابع بسته نشوند و یا در استثنائات مهم، پنهان گردند. چنین وضعیتی احتمالاً به انکار خدمات منتهی می‌شود. شایان ذکر است، عدم استفاده از یک دستور *try-with-resources* نمی‌تواند به‌خودی خود، یک آسیب‌پذیری امینتی محسوب شود، زیرا نوشتن گروهی که به‌درستی ساخته شده‌است، از بلاک‌های *try-catch-finally* تودرتو که از منابع در حال استفاده حفاظت می‌کنند، امکان‌پذیر است. با این‌حال، شکست در کنترل صحیح چنین خطاهایی، منبع شایع آسیب‌پذیری‌ها است. استفاده از یک دستور *try-with-resources*، این مشکل را با این تضمین که منابع به‌درستی مدیریت می‌شوند و استثنائات هیچ‌گاه پنهان نمی‌گردند، از بین خواهد برد.

۸,۴ از اعلان برای تأیید عدم وجود خطاهای زمان اجرا، استفاده نکنید

تست‌های تشخیص می‌توانند با استفاده از دستور *assert* در برنامه‌ها گنجانده شوند. اساساً هدف از اعلان‌ها، استفاده در طول اشکال‌زدایی است و اغلب، پیش از آن‌که کد، با استفاده از سوئیچ زمان اجرای *-disableassertions* یا (*-da*) مستقر شود، خاموش می‌گردند. در نتیجه، اعلانات باید برای حفاظت از فرض‌های نادرست مربوط به برنامه‌نویس به کار روند و نباید از آنها برای بررسی خطاهای زمان اجرا استفاده نمود.

اعلانات هیچ‌گاه نباید برای تأیید عدم وجود خطاهای زمان اجرا (در برابر منطق) استفاده شوند. خطاهایی از قبیل:

- ورود کاربری نامعتبر (شامل پارامترهای خط فرمان و متغیرهای محیطی)
- خطاهای فایل (مانند خطاهای مربوط به بازکردن، خواندن، یا نوشتن فایل‌ها)
- خطاهای شبکه (از جمله خطاهای پروتکل شبکه) شرایط کمبود یا نبود حافظه (وقتی‌که ماشین مجازی جاوا نمی‌تواند به یک شی جدید، فضا اختصاص دهد و زباله‌روب نمی‌تواند فضای کافی را قابل دسترسی سازد).
- استفاده یا مصرف کامل منابع سیستم (مانند توصیفگران خارج از فایل، پردازش‌ها، و نخ‌ها)

- خطاهای فراخوانی سیستم (مثل خطاهای مربوط به اجرای فایل‌ها، قفل کردن، یا از قفل در آورن میوتکس‌ها[۱۴۸])
- مجوزهای نامعتبر (مانند فایل، حافظه، و کاربر)

کدی که از یک خطای I/O محافظت می‌کند، نمی‌تواند به عنوان یک اعلان پیاده‌سازی شود، زیرا باید در قابل اجرایی مستقر، حاضر باشد.

اعلانات عموماً برای برنامه‌های سرور یا سیستم تعبیه‌شده در استقرا، نامناسب هستند. یک اعلان ناموفق می‌تواند به حمله‌ی DoS منجر شود. اگر توسط یک مهاجم اعمال شود، یک مُد شکست نرم، همچون نوشتن در یک فایل گزارش، مناسب‌تر است.

۱.۸.۴ نمونه‌کد ناسازگار

این نمونه، از دستور *assert* برای تأیید در دسترس بودن ورودی استفاده می‌کند.

```
BufferedReader br;
// Set up the BufferedReader br
String line;
// ...
line = br.readLine();
assert line != null;
```

از آن‌جایی‌که در دسترس بودن ورودی به کاربر بستگی دارد و می‌تواند در هر زمان، در طول اجرای برنامه به‌طور کامل استفاده شود، یک برنامه‌ی دقیق باید برای مدیریت صحیح، آمادگی داشته‌باشد و از حالت در دسترس نبودن ورودی، بازیابی گردد. با این‌حال، استفاده از دستور *assert* برای تأیید این‌که برخی از ورودی‌های مهم در دسترس هستند، به دلیل آن‌که می‌تواند منجر به پایان ناگهانی فرایند شود، که خود در نهایت به DoS منتهی می‌شود، نادرست است.

۲.۸.۴ راه‌حل سازگار

این راه‌حل، روش توصیه‌شده برای شناسایی و مدیریت در دسترس نبودن ورودی را نشان می‌دهد.

```
BufferedReader br;
// Set up the BufferedReader br
String line;
// ...
line = br.readLine();
if (line == null)
{
	// Handle error
}
```

۳.۸.۴ کاربرد

اعلانات، ابزار تشخیص ارزشمندی برای یافتن و حذف نواقص نرم‌افزاری است که ممکن است به آسیب‌پذیری منتهی شوند. عدم وجود اعلانات، به این معنا نیست که کد، عاری از اشکال است.

در مجموع، استفاده‌ی ناصحیح از دستور *assert* برای بررسی در زمان اجرا به جای بررسی خطاهای منطقی، نمی‌تواند به‌صورت خودکار قابل کشف و شناسایی باشد.

۹,۴ از همان نوع عملوندهای دوم و سوم برای عبارت‌های شرطی استفاده کنید

عملگر شرطی :?، از مقدار بولین مربوط به عملوند اول، برای تعیین این‌که کدام یک از دو عبارت دیگر ارزیابی خواهند شد، استفاده می‌کند. شکل کلی یک عبارت شرطی جاوا، به‌صورت زیر است:

```
operand1 ? operand2 : operand3
```

- اگر مقدار عملگر اول، (*operand1*) صحیح (*true*) باشد، عبارت عملگر دوم (*operand2*) انتخاب می‌شود.
- اگر مقدار عملگر اول، *fulse* باشد، عبارت عملوند سوم (*operand3*) انتخاب می‌شود.

عملگر شرطی، از نظر نحوی، راست‌محور است (اولویت عملیات موجود در سمت راست بیشتر است). برای مثال، *a?b:c?d:e?f:g* معادل *a?b:(c?d:(e?f:g))* است.

قواعد JLS برای تعیین نوع نهایی یک عبارت شرطی (جدول ۴-۱) دشوار و پیچیده هستند، به‌نحوی‌که برنامه‌نویسان، از تبدیلات نوعی مورد نیاز برای اصلاحاتی که نوشته‌اند، غافل‌گیر خواهند شد.

جدول ۴-۱: تعیین نوع نتیجه‌ی یک عبارت شرطی

قاعده	Operand2	Operand3	نوع نتیجه
۱	Type T	Type T	Type T
۲	boolean	Boolean	boolean
۳	Boilean	boolean	boolean
۴	null	reference	reference
۵	reference	null	reference
۶	Byte یا byte	Short یا short	short

۷	Short یا short	Byte یا byte	short
۸	Byte، char، short، byte Character، Short	constant int	char، short، byte، اگر مقدار int قابل نمایش باشد
۹	constant int	Byte، char، short، byte Character، Short	char، short، byte، اگر مقدار int قابل نمایش باشد
۱۰	سایر اعداد	سایر اعداد	نوع ارتقایافته‌ی عملوندهای دوم و سوم
۱۱	T1 = boxing conversion (S1)	T2 = boxing conversion (S2)	تبدیل ضبط را به حداقل کران بالای T1 و T2 به کار برید

تعیین نوع نهایی، از بالای جدول شروع می‌شود؛ کامپایلر، قاعده‌ی تطبیق ابتدایی را اعمال می‌کند. ستون‌های عملوند Operand 2 و Operand 3، به‌ترتیب، به *operand2* و *operand3* اشاره می‌کنند. در جدول، *constant int*، به عبارات ثابت از نوع *int* (همچون "0" یا متغیرهای اعلان‌شده‌ی *final*) مربوط می‌شود. در ردیف پایانی جدول، *S1* و *S2*، به‌ترتیب، انواع عملوندهای دوم و سوم هستند. *T1* و *T2*، به‌ترتیب، انواعی هستند که از اعمال تبدیل باکسینگ به *S1* و *S2* نتیجه می‌شوند. نوع عبارت شرطی، از اعمال کمترین کران بالای *T1* و *T2* حاصل می‌شود.

پیچیدگی قواعدی که نوع نهایی یک عبارت شرطی را تعیین می‌کنند، می‌توانند منجر به تبدیل‌های نوعی ناخواسته شوند. در نتیجه، عملوندهای دوم و سوم هر عبارت شرطی باید از انواع مشابهی برخوردار باشند. این توصیه در مورد عناصر اصلی باکس‌شده نیز صادق است.

۱.۹.۴ نمونه‌کد ناسازگار

در این نمونه، برنامه‌نویس انتظار دارد هر دو دستور چاپ، مقدار *alpha* را به‌عنوان *char* چاپ کنند.

```
public class Expr {
	public static void main(String[] args) {
		char alpha = 'A';
		int i = 0;
		// Other code. Value of i may change
		boolean trueExp = true; // Expression that evaluates to true
		System.out.print(trueExp ? alpha : 0); // Prints A
		System.out.print(trueExp ? alpha : i); // Prints 65
	}
}
```

دستور چاپ اول، *A* را چاپ خواهد کرد، زیرا کامپایلر، قاعده‌ی ۸ را از جدول ۴-۱ جهت تعیین این موضوع که عملوندهای دوم و سوم مربوط به عبارت شرطی، از نوع *char* هستند یا به آن تبدیل می‌شوند، اعمال می‌کند. با این‌حال، دستور چاپ دوم، ۶۵ (مقدار *alpha* به‌عنوان یک *int*) را چاپ می‌کند. اولین قاعده‌ی تطبیقی، قاعده‌ی ۱۰ جدول است. در نتیجه، کامپایلر مقدار *alpha* را به نوع *int* ارتقا می‌دهد.

۲.۹.۴ راه‌حل سازگار

این راه‌حل، از انواع مشابه برای عملوندهای دوم و سوم مربوط به هر عبارت شرطی، استفاده می‌کند. تبدیلات نوع صریح, نوع موردانتظار برنامه‌نویس را تعیین می کند.

```
public class Expr {
	public static void main(String[] args) {
		char alpha = 'A';
		int i = 0;
		boolean trueExp = true; // Expression that evaluates to true
		System.out.print(trueExp ? alpha : 0); // Prints A
		// Deliberate narrowing cast of i; possible truncation OK
		System.out.print(trueExp ? alpha : ((char) i)); // Prints A
	}
}
```

وقتی مقدار *i* در عبارت شرطی دوم خارج از محدوده‌ای قرار می‌گیرد که می‌تواند به‌عنوان *char* نمایش داده‌شود، تبدیل نوع صریح، مقدار آن را کاهش خواهد داد. این کاربرد، با استثنائات موسوم به *NUM12-EX0* از *NUM12-J* هماهنگی دارد. مطمئن شوید که تبدیلات انواع عددی به انواع باریک‌تر، منجر به از دست رفتن یا تفسیر اشتباه داده‌ها نشوند.

۳.۹.۴ نمونه‌کد ناسازگار

این نمونه، ۱۰۰ را به‌عنوان اندازه‌ی *HashSet*، به جای نتیجه‌ی موردانتظار (مقدار بین ۰ و ۵۰) چاپ خواهد کرد. ترکیب مقادیر انواع *short* و *int* در پارامتر دوم عبارت شرطی (عمل *i*-1)، باعث می‌شود که

نتیجه، یک *int* باشد (همان‌طور که توسط قواعد ارتقای *integer* تعیین شده‌اند). در نتیجه، شی *short* در پارامتر سوم، به‌عنوان یک *short* از باکس خارج می‌شود و سپس، به یک *int* ارتقا می‌یابد. بنابراین، نتیجه‌ی

```
public class ShortSet {
	public static void main(String[] args) {
		HashSet<Short> s = new HashSet<Short>();
		for (short i = 0; i < 100; i++) {
			s.add(i);
			// Cast of i-1 is safe, because value is always representable
			Short workingVal = (short) (i-1);
			// ... Other code may update workingVal
			s.remove(((i % 2) == 1) ? i-1 : workingVal);
		}
		System.out.println(s.size());
	}
}
```

۴.۹.۴ راه‌حل سازگار

این راه‌حل، عملوند دوم را به نوع *short*، تبدیل می‌کند و سپس، متد *short.valueof()* را برای ایجاد نمونه‌ای که مقدار آن *i*-1 است، فراخوانی می‌نماید:

```
public class ShortSet {
	public static void main(String[] args) {
		HashSet<Short> s = new HashSet<Short>();
		for (short i = 0; i < 100; i++) {
			s.add(i);
			// Cast of i-1 is safe, because the
			// resulting value is always representable
			Short workingVal = (short) (i-1);
			// ... other code may update workingVal Cast of i-1 is safe,
			// because the resulting value is always representable
			s.remove(((i % 2) == 1) ? Short.valueOf((short) (i-1)) :
			workingVal);
		}
		System.out.println(s.size());
	}
}
```

در نتیجه‌ی تبدیل، هر دو عملوند دوم و سوم عبارت شرطی، از نوع *short* هستند. فراخوانی *remove()*، نتیجه‌ی موردانتظار را ارائه خواهد داد.

نوشتن عبارات شرطی، از قبیل *((i % 2) == 1) ? (short) (i-1)) : workingVal*، با این دستورالعمل هماهنگی دارند، زیرا عملوندهای دوم و سوم در این فرم، هر دو نوع *short* را دارند. با این‌حال، این گزینه کارایی کمتری دارد، زیرا از باکس خارج کردن *workingVal* را روی هر تکرار حلقه، و باکس نمودن خودکار نتیجه‌ی عبارت شرطی (از *short* به *short*) را روی هر تکرار حلقه، بالاجبار، اعمال خواهند شد.

۵,۹,۴ کاربرد

وقتی عملوندهای دوم و سوم یک عبارت شرطی، انواع مختلفی داشته‌باشند، می‌توانند در معرض تبدیلات نوعی غیرمنتظره باشند. کشف و شناسایی خودکار عبارات شرطی، که انواع عملوندهای دوم و سوم آنها متفاوت بوده، صریح و آشکار است.

۱۰,۴ کنترل‌های مستقیم منابع سیستم را سریال نکنید

اشیای متوالی می‌توانند ناحیه‌ی خارجی هر برنامه جاوا را تغییر دهند. آنها از مکانیزم‌های حفاظت‌شده مانند مهر و امضا کردن استفاده می‌کنند. اگر یک شی به یکی از منابع سریال‌شده‌ی سیستم اشاره کند، یک مهاجم خواهد توانست توالی شی را تغییر دهد. حتی ممکن است بتواند مدیریت سریالی منابع را نیز دچار تغییر نماید. برای مثال، یک مهاجم می‌تواند مدیر یک فایل سریال‌شده را تغییر دهد. در نتیجه، وی قادر خواهد بود با استفاده از مدیر فایل، هرگونه عملیاتی را با استفاده از مسیر و نام فایل انجام دهد.

۱,۱۰,۴ نمونه کد ناسازگار

این مثال، شی سریال‌پذیر *File* را در کلاس *Ser* اعلان می‌کند:

```
final class Ser implements Serializable {
	File f;
	public Ser() throws FileNotFoundException {
		f = new File("c:\\filepath\\filename");
	}
}
```

فرم سریال‌شده‌ی شی نشان می‌دهد که مسیر فایل می‌تواند تغییر کند. هنگامی‌که شی از حالت سریال خارج می‌شود، عملیات ترجیح می‌دهد از مسیر تغییریافته استفاده نماید که می‌تواند موجب خوانده شدن و یا تغییر فایل اشتباه شود.

۲,۱۰,۴ راه‌حل سازگار (عدم پیاده‌سازی سریال‌پذیری)

این راه‌حل، یک کلاس *final* با نام *Ser* را نشان می‌دهد که *java.io.Serializable* را پیاده‌سازی نکرده‌است. درنتیجه، شی *File* نمی‌تواند سریال شود.

```
final class Ser {
	File f;
	public Ser() throws FileNotFoundException {
		f = new File("c:\\filepath\\filename");
	}
}
```

۳،۱۰،۴راه‌حل سازگار (شی *Transient*)

این راه‌حل، شی *Transient* از *File* را اعلان می‌کند. مسیر فایل توسط باقی کلاس، سریال نمی‌شود. در نتیجه، در معرض دید مهاجمان قرار نخواهد گرفت.

```
final class Ser implements Serializable {
	transient File f;
	public Ser() throws FileNotFoundException {
		f = new File("c:\\filepath\\filename");
	}
}
```

۴،۱۰،۴کاربرد

مدیریت مستقیم از سریال خارج نمودن منابع یک سیستم، می‌تواند منجر به تغییر منابع اشاره‌شده شود.

۱۱,۴ استفاده از تکرارکننده‌ها را به شمارنده‌ها، ترجیح دهید

طبق مستند واسط Enumeration<E> متعلق به API جاوا، یک شی که واسط *Enumeration* را پیاده‌سازی می‌کند، یک‌سری از عناصر را یک‌به‌یک تولید می‌نماید. فراخوانی‌های موفق، متد *nextElement* عناصر ایجادشده‌ی موفق را باز می‌گرداند. به‌عنوان مثال، کد زیر از یک *Enumeration* برای نمایش محتوای یک *Vector* استفاده می‌کند:

```
for (Enumeration e = vector.elements(); e.hasMoreElements();)
{
      System.out.println(e.nextElement());
}
```

API جاوا توصیه می‌کند "پیاده‌سازی‌های جدید باید *Itrator* را در کارایی *Enumeration* درنظر بگیرند". تکرارکننده‌ها، برتر از شمارنده‌ها هستند، زیرا آنها از نام‌گذاری ساده‌تری برای متدها استفاده می‌کنند؛ همچنین، به هنگام حذف عناصر یک مجموعه، در حالی‌که در طول مجموعه تکرار می‌گردند، از نظر نحوی، بهتر تعریف شده‌اند. در نتیجه، در بررسی مجموعه‌های تکراری، باید تکرارکننده‌ها به شمارنده‌ها ترجیح داده‌شوند.

۱,۱۱,۴ نمونه‌کد ناسازگار

این مثال، یک کلاس *BankOperations* را به‌همراه یک متد *removeAccount()*، که در راستای پایان دادن به تمامی دارندگان حساب‌های خاصی که با نام، شناسایی می‌شوند، استفاده می‌شود، پیاده‌سازی می‌کند. در صورتی‌که شخصی بیش از یک حساب داشته‌باشد، می‌توان اسامی را در یک بردار تکرار نمود.. متد *remove()* تلاش دارد تا تمامی ورودی‌های بردار را تکرار نماید. هر ورودی را با نام "Harry," مقایسه می‌کند.

با مواجه شدن با اولین "Harry," ، ورودی با موفقیت حذف می‌گردد، و اندازه‌ی ابعاد بردار، به درخت کاهش می‌یابد. با این‌حال، اندیس *Enumeration* همچنان بدون تغییر باقی می‌ماند، زیرا برنامه ترجیح می‌دهد که بعداً (در پایان)، مقایسه با "Tom." انجام گیرد. در نتیجه، دومین "Harry." در بردار، ناامیدانه باقی می‌ماند و به موقعیت دوم بردار منتقل می‌شود.

```
class BankOperations {
	private static void removeAccounts(Vector v, String name) {
		Enumeration e = v.elements();
		while (e.hasMoreElements()) {
			String s = (String) e.nextElement();
			if (s.equals(name)) {
				v.remove(name); // Second Harry is not removed
			}
		}
		// Display current account holders
		System.out.println("The names are:");
		e = v.elements();
		while (e.hasMoreElements()) {
			// Prints Dick, Harry, Tom
			System.out.println(e.nextElement());
		}
	}
	public static void main(String args[]) {
		// List contains a sorted array of account holder names
		// Repeats are admissible
		List list = new ArrayList(Arrays.asList(
			new String[] {"Dick", "Harry", "Harry", "Tom"}));
		Vector v = new Vector(list);
		removeAccount(v, "Harry");
	}
}
```

۲.۱۱.۴ راه‌حل سازگار

طبق مستند واسط *Iterator<E>* متعلق به API جاوا، *Iterator* به دو روش در *Enumeration*، در چارچوب مجموعه‌ی جاوا ایفای نقش می‌نماید:

- تکرارکننده‌ها به فراخواننده‌ها اجازه می‌دهند تا عناصر اصلی مجموعه را در صورت تکرار، با نحو خوش‌تعریف حذف نمایند.
- اسامی متدها، بهبود خواهد یافت.

این راه‌حل، راه‌حل مسئله‌ی مطرح‌شده در نمونه‌کد ناسازگار است و مزایای استفاده از *Iterator* را در *Enumeration* مشخص می‌کند.

```
class BankOperations {
	private static void removeAccounts(Vector v, String name) {
		Iterator i = v.iterator();
		while (i.hasNext()) {
			String s = (String) i.next();
			if (s.equals(name)) {
				i.remove(); // Correctly removes all instances
				// of the name Harry
			}
		}
		// Display current account holders
		System.out.println("The names are:");
		i = v.iterator();
		while (i.hasNext()) {
			System.out.println(i.next()); // Prints Dick, Tom only
		}
	}
	public static void main(String args[]) {
		List list = new ArrayList(Arrays.asList(
			new String[] {"Dick", "Harry", "Harry", "Tom"}));
		Vector v = new Vector(list);
		remove(v, "Harry");
	}
}
```

۳,۱۱,۴ کاربرد

ممکن است استفاده از *Enumeration* در زمانی‌که ترجیح می‌دهید عملگرها را در یک مجموعه‌ی پرتکرار حذف کنید، موجب پیدایش رفتار غیرمنتظره‌ای در برنامه شود.

۱۲,۴ از بافرهای مستقیم برای اشیای کوتاه‌عمر غیرمتناوب استفاده نکنید

کلاس‌های جدید I/O (NIO) در *java.nio* امکان ایجاد و استفاده از بافرهای مستقیم را فراهم می‌آورند. این بافرها، به‌صورت مهیبی موجب افزایش فعالیت‌های ورودی و خروجی تکراری از این طریق می‌شوند. اگرچه، هزینه‌ی ایجاد و احیای آنها بیشتر از ایجاد و احیای بافرهای غیرمستقیم مبتنی بر هیپ است، زیرا بافرهای مستقیم با استفاده از کدهای محلی مختص سیستم‌عامل مدیریت می‌شوند. این امر، هزینه‌ی مدیریت را نیز می‌افزاید. از این‌رو، بافرهای مستقیم، یک انتخاب ضعیف برای موارد استفاده‌ی نامتناوب محسوب می‌شوند. بافرهای مستقیم، همواره خارج از محدوده‌ی زباله‌روب جاوا هستند. در نتیجه، ناعاقلانه است که از بافرهای مستقیم استفاده کنیم و این موضوع می‌تواند موجب نشت حافظه شود. در پایان، تخصیص متناوب بافرهای مستقیم و بزرگ می‌تواند موجب خطای *OutOfMemoryError* گردد.

۱,۱۲,۴ نمونه‌کد ناسازگار

در این مثال، از شیای محلی کوتاه‌عمر و درازعمر، به‌ترتیب، با نام‌های *rarelyUsed-Buffer* و *heavilyUsedBuffer*، استفاده شده‌است. هر دو، در حافظه‌ی غیر هیپ اختصاص داده شده‌اند. هیچ‌یک از آنها زباله‌روب نیست.

```
ByteBuffer rarelyUsedBuffer = ByteBuffer.allocateDirect(8192);
// Use rarelyUsedBuffer once
ByteBuffer heavilyUsedBuffer = ByteBuffer.allocateDirect(8192);
// Use heavilyUsedBuffer many times
```

۲,۱۲,۴ راه‌حل سازگار

این راه‌حل سازگار، از یک بافر غیرمستقیم برای تخصیص شی کوتاه‌عمر غیرمتناوب استفاده می‌کند. بافری که بسیار مورد استفاده قرار گرفته‌است، به‌طور مناسب ادامه می‌دهد تا از یک بافر مستقیم غیرهیپ غیرزباله‌روب استفاده نماید.

```
ByteBuffer rarelyUsedBuffer = ByteBuffer.allocate(8192);
// Use rarelyUsedBuffer once
ByteBuffer heavilyUsedBuffer = ByteBuffer.allocateDirect(8192);
// Use heavilyUsedBuffer many times
```

۳.۱۲.۴ کاربرد

بافرهای مستقیم در حوزه‌ی زباله‌روب جاوا قرار دارند. در صورتی‌که ناعاقلانه استفاده شوند، می‌توانند موجب نشت حافظه گردند. به‌طور کلی، بافرهای مستقیم باید هنگامی‌که استفاده از آنها منجر به افزایش قابل توجه عملکرد می‌شود، اختصاص یابند.

۱۳,۴ اشیای کوتاه‌عمر را از اشیای نگهدارنده‌ی درازعمر، حذف کنید

همواره، هنگامی‌که وظایف به اتمام می‌رسند، اشیای کوتاه‌عمر را از اشیای نگهدارنده‌ی درازعمر، حذف کنید. برای مثال، اشیایی که به شی *java.nio.channels.SelectionKey* پیوست شده‌اند باید زمانی‌که برای مدت کوتاهی مورد نیاز هستند، حذف گردند. انجام این‌کار، احتمال نشت حافظه را کاهش می‌دهد. به‌طور مشابه، استفاده از داده‌های مبتنی بر ساختمان داده، نظیر *ArrayList*، می‌تواند نیازی را معرفی کند تا عدم حضور یک ورودی را به‌صراحت، با تنظیم عناصر آرایه‌ی *ArrayList* به مقدار *null*، نشان دهد.

در این راهنما، آدرس‌های اشیای خاص، به نگهدارندگان آنها اشاره دارند. برای مثال، حذف اشیا به جمع‌آوری زباله کمک نمی‌کند.

۱,۱۳,۴ نمونه‌کد ناسازگار (حذف اشیای کوتاه‌عمر)

در این مثال، یک آرایه درازعمر به نام *ArrayList*، به هر دو نوع عناصر کوتاه و درازعمر اشاره دارد. برنامه‌نویس، با استفاده از پرچم "*dead*"، روی عناصر غیرمرتبط، علامت‌گذاری می‌کند.

```
class DataElement {
	private boolean dead = false;
	// Other fields
	public boolean isDead() { return dead; }
	public void killMe() { dead = true; }
}
// ... Elsewhere
List<DataElement> longLivedList = new ArrayList<DataElement>();
// Processing that renders an element irrelevant
// Kill the element that is now irrelevant
longLivedList.get(someIndex).killMe();
```

زباله‌روب نمی‌تواند شی *DataElement* مرده را، تا زمانی‌که مرجع‌دهی نشود، جمع‌آوری نماید. به‌یاد داشته‌باشید که همه‌ی متدهایی که می‌توانند روی اشیای کلاس *DataElement* عمل کنند، باید بررسی نمایند که آیا مورد در دست، مرده است یا خیر.

۲,۱۳,۴راه‌حل سازگار (تنظیم مرجع به *null*)

در این راه‌حل، به جای آن‌که از یک پرچم *dead* استفاده شود، برنامه‌نویس مقدار *null* را به عناصر *ArrayList* غیرمرتبط نسبت می‌دهد.

```
class DataElement {
	// Dead flag removed
	// Other fields
}
// Elsewhere
List<DataElement> longLivedList = new ArrayList<DataElement>();
// Processing that renders an element irrelevant
// Set the reference to the irrelevant DataElement to null
longLivedList.set(someIndex, null);
```

توجه داشته‌باشید، تمامی کدهایی که روی *LongLivedList* عمل می‌کنند، اکنون باید ورودی‌های لیست *null* را بررسی نمایند.

۳,۱۳,۴راه‌حل سازگار (استفاده از الگوی شی *Null*)

این راه‌حل، با استفاده از یک شی نگهبان، از مسائل مربوط به مرجع مقادیر *null* اجتناب می‌کند. این تکنیک، به‌عنوان الگوی شی *Null* شناخته می‌شود (به الگوی نگهبان نیز شهره است).

```
class DataElement {
	public static final DataElement NULL = createSentinel();
	// Dead flag removed Other fields
	private static final DataElement createSentinel() {
		// Allocate a sentinel object, setting all its fields
		// to carefully chosen "do nothing" values
	}
}
// Elsewhere
List<DataElement> longLivedList = new ArrayList<DataElement>();
// Processing that renders an element irrelevant
// Set the reference to the irrelevant DataElement to the NULL object
longLivedList.set(someIndex, DataElement.NULL);
```

در صورت امکان، برنامه‌نویس باید این الگوی طراحی را در برابر مرجع صریح مقادیر *null* انتخاب کند.

هنگامی‌که از این الگو استفاده می‌کنید، شی *null* باید تنها و از نوع *final* باشد. ممکن است *public* یا *private* باشد؛ البته به طراحی کلاس *DataElement* بستگی دارد. وضعیت شی *null* باید پس از ایجاد، تغییرناپذیر باشد. تغییرناپذیری می‌تواند با استفاده از فیلدهای *final* و یا با استفاده از کد صریح در متدهای کلاس *DataElement*، اعمال گردد.

۴.۱۳.۴ کاربرد

رهاکردن اشیای کوتاه‌عمر در اشیای نگهدارنده‌ی درازعمر، ممکن است موجب مصرف حافظه شود که در این‌صورت، نمی‌تواند توسط زباله‌روب بازیابی گردد. این امر، موجب خستگی حافظه و ممنوعیت خدمات ممکن می‌شود.

فصل پنجم: قابل درک بودن برنامه

منظور از قابل درک بودن برنامه، فهمیدن ساده‌ی آن برنامه است. در واقع، باید از طریق خواندن کد منبع، به‌راحتی بتوان تعریف یک برنامه، کاری که انجام می‌دهد، و این‌که چگونه کار می‌کند، را فهمید. نگهداری کد قابل فهم، ساده‌تر است، زیرا احتمال این‌که نگهدارنده‌ی نرم‌افزار نقص‌ها را برای کد واضح و قابل درک نمایان سازد، کمتر است. قابل درک بودن، به تحلیل‌های کتابچه‌ی راهنمای کد منبع نیز کمک می‌کند، زیرا این اجازه را به مخاطب تا که به‌سادگی، آسیب‌پذیری‌ها و نقاط نقص را دریابد.

برخی از دستورالعمل‌های این فصل، به‌صورت ذهنی است: این دستورالعمل‌ها به برنامه‌نویس جاوا کمک می‌کنند تا کد خوانا و واضح‌تری بنویسد. عدم پیروی از این دستورالعمل‌ها ممکن است منجر به تولید کدهای مبهم و طراحی‌های دارای نقص باشد.

1.5 در استفاده از شناسه‌های بصری گمراه‌کننده و الفاظ، دقت کنید

از شناسه‌های بصری مجزا استفاده کنید تا احتمال کمتری برای نخواندن در طی تولید و بازبینی کد وجود داشته‌باشد. بسته به فونتی که استفاده می‌کنید، کاراکترهای مطمئن، از نظر بصری، شبیه به‌هم هستند یا حتی یکسان هستند و ممکن است اشتباه تفسیر شوند. مثال جدول ۵-۱ را در نظر بگیرید.

جدول ۵-۱: کاراکترهای گمراه‌کننده

کاراکتر موردنظر	ممکن است با این کاراکتر اشتباه گرفته‌شود و بالعکس
0 (صفر)	O (O بزرگ) D (D بزرگ)
1 (یک)	I (I بزرگ) l (L کوچک)
2 (دو)	Z (Z بزرگ)
5 (پنج)	S (S بزرگ)
8 (هشت)	B (B بزرگ)
n (N کوچک)	h (H کوچک)
rn (R کوچک، N کوچک)	m (M کوچک)

یکی از ملزومات مشخصات زبان جاوا این است که کد منبع برنامه، با استفاده از کدگذاری کاراکترهای یونیکد (یونیکد ۲۰۱۳) نوشته شود. بعضی از کاراکترهای یونیکد متمایز، هنگام نمایش در بسیاری از فونت‌های معمول، نمایان‌گر یک علامت حجاری‌شده (گلیف)[۱۴۹] هستند. برای مثال، کاراکترهای *Greek* و *Coptic* (محدوده‌ی یونیکد بین 0370-03FF)، متناوباً زیرمجموعه‌ی کاراکترهای *Greek* از نمادهای الفبایی ریاضیاتی قابل تشخیص (محدوده‌ی یونیکد 1D400-1D7FF) نیستند.

[۱۴۹] Glyph

از تعریف کردن شناسه‌هایی که کاراکترهای یونیکد با سربار گلیف دارند، اجتناب کنید. یک روش ساده، استفاده از کاراکترهای *ASCII* یا کاراکتر های لاتین است. توجه داشته‌باشید که کاراکترهای *ASCII* زیرمجموعه‌ای از کاراکترهای یونیکد هستند.

از چندین شناسه، که با یک یا چند کاراکتر مشابه بصری، قابل تشخیص هستند، استفاده نکنید. همچنین، بخش‌های ابتدایی شناسه‌های بلند، به تشخیص کمک می‌کنند.

یک عدد صحیح لفظی، از نوع *long* است اگر حروف *ASCII*، *L* یا *l*، پسوند آن قرار گیرند. این امر، اغلب برای تشخیص نوع *int*، دشوار است. پسوند *L* ترجیح داده می‌شود، زیرا حرف *l* (حرف *L* کوچک) ممکن است با عدد دجیتالی 1 اشتباه گرفته‌شود. در نتیجه، هنگامی‌که برنامه‌نویس قصد دارد عدد صحیح را از نوع *long* تمییز دهد، از *L* برای روشن‌سازی هدف وی استفاده کنید.

الفاظ نوع صحیح که با صفر شروع می‌شوند، در حقیقت، یادآور مقادیر برمبنای ۸ هستند، نه مقادیر دهدهی. یک عدد بر مبنای ۸ می‌تواند اعداد مثبت، صفر، و یا منفی را نشان دهد. این سوءتفاهم ممکن است موجب بروز اشتباهات برنامه‌نویسی و نیز، تعریف چندین ثابت و سعی در راستای ارتقای فرمت توسط صفر، خواهد شد.

۱,۱,۵ نمونه‌کد ناسازگار

این نمونه‌کد، دارای دو متغیر به نام‌های *stem* و *stern* و با محدوده‌های یکسان است که می‌تواند به‌سادگی موجب سردرگمی و تعویض تصادفی گردد.

```
int stem; // Position near the front of the boat
/* ... */
int stern; // Position near the back of the boa
```

۲,۱,۵ راه‌حل سازگار

این راه‌حل، با اضافه کردن شناسه‌های بصری به متغیرها، این سردرگمی را از بین می‌برد:

```
int bow; // Position near the front of the boat
/* ... */
int stern; // Position near the back of the boat
```

۳.۱.۵ نمونه‌کد ناسازگار

این مثال، نتیجه‌ی حاصل‌جمع یک *int* و یک *long* را چاپ می‌کند. اگرچه، ممکن است به‌نظر برسد که دو عدد صحیح (11111) در حال جمع‌شدن هستند.

```
public class Visual
{
	public static void main(String[] args)
	{
		System.out.println(11111 + 1111l);
	}
}
```

۴.۱.۵ راه‌حل سازگار

در این راه‌حل، از حرف بزرگ *L* (*long*) به جای حرف کوچک *l*، برای متفاوت ساختن ظاهر بصری عدد صحیح دوم استفاده شده‌است. رفتار آن مانند نمونه‌کد ناسازگار است، اما هدف برنامه نویس، روشن است:

```
public class Visual
{
	public static void main(String[] args)
	{
		System.out.println(11111 + 1111L);
	}
}
```

۵.۱.۵ نمونه‌کد ناسازگار

این مثال، هنگامی‌که اعداد را در آرایه ذخیره می‌کند، مقادیر عددی دهدهی را با مقادیر برمبنای ۸، تلفیق می‌نماید:

```
int[] array = new int[3];
void exampleFunction()
{
	array[0] = 2719;
	array[1] = 4435;
	array[2] = 0042;
	// ...
}
```

این کد، نشان می‌دهد که عنصر سوم در *array*، مقدار ۴۲ دهدهی را درنظر دارد. درحالی‌که، مقدار دهدهی ۳۴ (عدد متناظر ۴۲ در مبنای ۸) تخصیص داده می‌شود.

۶,۱,۵ راه حل سازگار

هنگامی‌که الفاظ اعداد صحیح برای نمایش مقادیر دهدهی، موردنظر هستند، از قرار دادن صفرها پیش از عدد اجتناب کرده و از تکنیک دیگری استفاده کنید: برای مثال، از ایجاد فضای سفید (خالی) برای حفظ هماهنگی اعداد، استفاده نمایید.

```
int[] array = new int[3];
void exampleFunction()
{
	array[0] = 2719;
	array[1] = 4435;
	array[2] =	42;
	// ...
}
```

۷,۱,۵ کاربرد

شکست در استفاده از شناسه‌های متمایز بصری می‌تواند نتیجه‌ی استفاده از شناسه‌های اشتباه باشد و منجر به رفتار غیرمنتظرانه‌ی برنامه گردد.

کشف شناسه‌ها با نام‌های بصری مشابه، ساده است. سردرگمی استفاده از l و عدد 1، هنگامی‌که عدد صحیح را به‌عنوان مقدار long نشان می‌دهد، می‌تواند منجر به محاسبات اشتباه شود. خودکارسازی تشخیص، امری بدیهی است.

ترکیب مقادیر دهدهی و مبنای ۸، می‌تواند منجر به مقداردهی اولیه یا تخصیص نادرست شود. تشخیص لیترال های عددی صحیح که با پیشوند صفر آغاز می شوند بدیهی است. اگرچه، تعیین اینکه برنامه‌نویس قصد استفاده از لفظ درمبنای ۸ دارد یا لفظ دهدهی، غیر قابل اجتناب است. براین اساس، تشخیص خودکار صدا، غیرقابل اجتناب است. ممکن است بررسی‌های هیورستیک، کاربردی باشد.

۲,۵ از بارگذاری مبهم متدهای متغیر *arity*، خودداری کنید

ویژگی متغیر *arity (varargs)*، در JDK v1.5.0 معرفی شد تا از متدهایی که تعداد متغیری از آرگومان‌ها را می‌پذیرند، پشتیبانی نماید. لازم است به‌ندرت و تنها، زمانی‌که مزایای آن کاملا مشخص و قانع‌کننده است، از این متدها استفاده کنید. به‌طور کلی، نباید یک متد *varargs* را بیش از حد، بارگذاری نمایید. در غیر این‌صورت، برای برنامه‌نویسان دشوار خواهد بود که نشان دهند کدام بارگذاری، بیش‌از حد فراخوانی شده‌است.

۱,۲,۵ نمونه‌کد ناسازگار

در این مثال، متد *overloading variable arity*، مشخص نکرده‌است که کدام تعریف *displayBooleans()* درخواست و احضار شده‌است:

```
class Varargs
{
	private static void displayBooleans(boolean... bool)
	{
		System.out.print("Number of arguments: "
			+ bool.length + ", Contents: ");
		for (boolean b : bool)
		{
			System.out.print("[" + b + "]");
		}
	}
	private static void displayBooleans(boolean bool1,boolean bool2)
	{
		System.out.println("Overloaded method invoked");
	}
	public static void main(String[] args)
	{
		displayBooleans(true, false);
	}
}
```

خروجی این برنامه، به‌صورت زیر خواهد بود، زیرا تعریف *nonvariable arity*، مشخص‌تر است و در نتیجه، سازگاری و تناسب بهتری برای آرگومان‌های فراهم‌شده دارد. با این‌حال، بهتر است از این پیچیدگی اجتناب شود.

```
Overloaded method invoked
```

۲,۲,۵ راه‌حل سازگار

همان‌گونه که در این راه‌حل مشاهده می‌شود، برای جلوگیری از متد *overloading variable arity*، از نامه‌های متد مشخص استفاده می‌شود تا اطمینان حاصل گردد که متد موردنظر، درخواست شده‌است:

```
class Varargs
{
	private static void displayManyBooleans(boolean... bool)
	{
		System.out.print("Number of arguments: "
			+ bool.length + ", Contents: ");
		for (boolean b : bool)
		{
			System.out.print("[" + b + "]");
		}
	}
	private static void displayTwoBooleans(boolean bool1,boolean bool2)
	{
		System.out.println("Overloaded method invoked");
		System.out.println("Contents: ["+ bool1 + "], [" + bool2 + "]");
	}
	public static void main(String[] args)
	{
		displayManyBooleans(true, false);
	}
}
```

۳,۲,۵ کاربرد

استفاده‌ی نادرست از متدهای *overloaded variable arity*، می‌تواند قابلیت خوانایی کد را مبهم و سخت سازد. نقض این قانون می‌تواند جهت افزایش بهره‌وری، مطلوب باشد. از جمله دلایل آن می‌توان به پیشگیری از ساخت و مقداردهی اولیه‌ی یک نمونه آرایه روی هر درخواست متد اشاره نمود.

```
public void foo() { }
public void foo(int a1) { }
public void foo(int a1, int a2, int... rest) { }
```

هنگامی‌که متدهای *variable arity* بارگذاری می‌شوند، لازم است از هرگونه ابهام ناشی از فراخوانی متد، جلوگیری به‌عمل آید. نمونه کد پیشن، از امکان انتخاب متد نادرست از طریق امضاهای متدهای نامبهم، جلوگیری می‌کند. تشخیص خودکار، ساده و آسان است.

۳,۵ جلوگیری و پیشگیری از شاخص‌های خطای *in-band*

یک شاخص خطای *in-band*، یک مقدار برگشتی از متدی است که نشان‌دهنده‌ی یک مقدار بازگشتی قانونی یا یک مقدار غیرقانونی است و خطایی را نشان می‌دهد. برخی مثال‌های رایج از شاخص‌های خطای *in-band*، شامل موارد زیر است:

- یک شی معتبر یا یک ارجاع تهی
- یک عدد صحیح نشان‌دهنده‌ی یک مقدار مثبت یا ۱-، که نشان می‌دهد یک خطا رخ داده‌است.
- آرایه‌ای از اشیای معتبر یا یک ارجاع تهی، که نشان‌دهنده‌ی عدم وجود اشیای معتبر است.

شاخص‌های خطای *in-band*، به فراخواننده برای بررسی خطا نیاز دارند. با این‌حال، این بررسی، اغلب نادیده گرفته می‌شود. عدم بررسی چنین شرایط خطایی، نه‌تنها CERT® Oracle® Secure Coding Standard for Java ™ [Long 2012] را مختل می‌کند ("EXP00-J. مقادیر بازگشتی توسط متدها را نادیده نگیرید")، بلکه اثر ناخوشایندی از انتشار مقادیر نامعتبر را در پی دارد، که بعدها ممکن است منجر به تهدیدی به‌عنوان یک مقدار معتبر در محاسبات بعدی شود.

از شاخص‌های خطای *in-band* استفاده نکنید. این شاخص‌ها بسیار کمتر از سایر زبان‌های برنامه‌نویسی، در کتابخانه‌ی هسته جاوا قرار دارند. با این‌وجود، در خانواده‌های *read(byte[] b, int off, int len)* و *read(char[] cbuf, int off, int len)* از متدهای موجود در java.io، استفاده می‌شوند.

بهترین راه برای نشان دادن یک وضعیت استثنایی در جاوا، ایجاد یک استثنا است نه برگشت یک کد خطا. استثنائات، برخلاف کدهای خطا، در سراسر حوزه‌ها و دامنه‌ها منتشر می‌شوند و نمی‌توانند به آسانی نادیده گرفته شوند. هنگام استفاده از استثنائات، کد تشخیص و راه‌انداز خطا، فارغ از جریان اصلی کنترل نگهداری می‌شوند.

۱,۳,۵ نمونه‌کد ناسازگار

این مثال تلاش می‌کند تا یک آرایه از کاراکترها را بخواند و بلافاصله، یک کاراکتر اضافی را پس از کاراکترهای خوانده‌شده، به بافر اضافه کند.

```
static final int MAX_READ = MAX - 1;
static final char TERMINATOR = '\\';
int read;
char [] chBuff = new char[MAX];
BufferedReader buffRdr;
// Set up buffRdr
read = buffRdr.read(chBuff, 0, MAX_READ);
chBuff[read] = TERMINATOR;
```

با این‌حال، اگر بافر ورودی در انتهای فایل، مقداردهی اولیه شود، متد *read()* مقدار ۱- را برمی‌گرداند و تلاش برای قراردادن کاراکتر پایانی، یک *ArrayIndexOutOfBoundsException* را ایجادخواهد کرد.

۲,۳,۵ راه‌حل سازگار (بسته‌بندی)

این راه‌حل، یک متد *readSafe()* را تعریف می‌کند که متد *read()* اصلی را بسته‌بندی می‌کند و در صورت یافتن پایان فایل، یک استثنا را ایجاد می‌نماید.

```
public static int readSafe(BufferedReader buffer, char[] cbuf,
int off, int len) throws IOException
{
	int read = buffer.read(cbuf, off, len);
	if (read == -1)
	{
		throw new EOFException();
	}
	else
	{
		return read;
	}
}
// ...
BufferedReader buffRdr;
// Set up buffRdr
try
{
	read = readSafe(buffRdr, chBuff, 0, MAX_READ);
	chBuff[read] = TERMINATOR;
}
catch (EOFException eof)
{
	chBuff[0] = TERMINATOR;
}
```

۳.۳.۵ کاربرد

استفاده از شاخص‌های خطای *in-band*، منجر به این موضوع می‌شود که برنامه‌نویس، کدهای وضعیت را بررسی نکند و یا از مقادیر بازگشتی نادرست استفاده نماید و در نتیجه، منجر به رفتار غیرمنتظره‌ای گردد.

باتوجه به نادربودن رخداد شاخص‌های خطای *in-band* در جاوا، کامپایل لیستی از همه‌ی متدهای کتابخانه استاندارد، که از آنها استفاده می‌نماید و نیز تشخیص خودکار استفاده از آنها، امکان‌پذیر است. با این‌حال، شناسایی و تشخیص استفاده‌ی ایمن از شاخص‌های خطای *in-band*، در موارد کلی و عمومی، امکان‌پذیر نیست.

بازگشت یک شی، که می‌تواند ناشی از شکست تهی باشد و یا به‌دلیل موفقیت معتبر صورت پذیرد، یک مثال رایج و معمول از یک شاخص خطای *in-band* است. اگرچه، غالبا طراحی‌های متد بهتری در دسترس هستند، بازگشت یک شی که می‌تواند تهی باشد، در بعضی شرایط قابل قبول است.

۴٫۵ تخصیص‌ها را در عبارات شرطی اجرا نکنید

استفاده‌ی مداوم از عملگرهای تخصیص در عبارات شرطی، نشان‌دهنده‌ی خطای برنامه‌نویس است و می‌تواند منجر به رفتار غیرمنتظره شود. عملگرهای تخصیص نباید در موارد زیر استفاده شوند:

- *If* (عبارت کنترلی)
- *While* (عبارت کنترلی)
- *do ... while* (عبارت کنترلی)
- *for* (عملوند دوم)
- *Switch* (عبارت کنترلی)
- *?:* (عملوند اول)
- && (عملوند یا)
- || (عملوند یا)
- *?:* (عملوند دوم یا سوم)، جایی‌که بیان سه‌گانه[۱۵۰] در هریک از این موارد استفاده می‌شود.

۱٫۴٫۵ نمونه‌کد ناسازگار

در این مثال، به جای آن‌که عبارت کنترلی *if* یک عبارت شرطی باشد، یک عبارت تخصیصی است.

```
public void f(boolean a, boolean b)
{
	if (a = b)
	{
		/* ... */
	}
}
```

اگرچه، قصد برنامه‌نویس می‌تواند تخصیص *b* به *a* و سپس، بررسی مقدار منتج باشد، این استفاده اغلب زمانی اتفاق می‌افتد که برنامه‌نویس به اشتباه از عملگر تخصیص "=" به جای عملگر برابری "==" استفاده می‌کند.

---

[۱۵۰] Ternary expression

۲,۴,۵ راه‌حل سازگار

دراین راه‌حل، بلوک شرطی، تنها، زمانی اجرا می‌شود که a برابر با b است. در واقع، تخصیص ناخواسته‌ی b به a نمی‌تواند اتفاق بیفتد.

```
public void f(boolean a, boolean b)
{
	if (a == b)
	{
		/* ... */
	}
}
```

۳,۴,۵ راه‌حل سازگار

هنگامی‌که تخصیص عمدی است، این راه‌حل، قصد برنامه‌نویس را روشن می‌سازد:

```
public void f(boolean a, boolean b)
{
	if ((a = b) == true
	{
		/* ... */
	}
}
```

۴,۴,۵ راه‌حل سازگار

واضح است که منطق را به‌عنوان یک تخصیص صریح و با شرط *if* بیان کنیم:

```
public void f(boolean a, boolean b)
{
	a = b;
	if (a)
	{
		/* ... */
	}
}
```

۵,۴,۵ نمونه‌کد ناسازگار

در این مثال، عبارت تخصیصی به‌عنوان عملگر عملوند "&&" ظاهر می‌شود.

```
public void f(boolean a, boolean b, boolean flag)
{
	while ( (a = b) && flag )
	{
		/* ... */
	}
}
```

به‌دلیل این‌که && یک عملگر مقایسه‌ای نیست، تخصیص یک عملوند، غیرقانونی است. مجددا، این مورد یکی از رایج‌ترین اشتباهات برنامه‌نویس است که از عملگر تخصیص "=" به جای عملگر برابری "==" استفاده می کند.

۶.۴.۵ راه‌حل سازگار

هنگام تخصیص ناخواسته‌ی b به a این بلوک شرطی، تنها هنگامی اجرا می‌شود که a برابر با b بوده و پرچم نیز true است.

```
public void f(boolean a, boolean b, boolean flag)
{
	while ( (a == b) && flag )
	{
		/* ... */
	}
}
```

۷.۴.۵ کاربرد

استفاده‌ی مداوم از عملگر تخصیص در عبارات شرطی کنترلی، نشان‌دهنده‌ی خطای برنامه‌نویس است و می‌تواند منجر به رفتار غیرمنتظره شود. به‌عنوان استثنایی برای این دستورالعمل، استفاده از عملگر تخصیص در عبارات شرطی هنگامی مجاز شده‌است که تخصیص، عبارت کنترلی نیست (یعنی تخصیص یک زیرعبارت است). این مورد را می‌توان در کد زیر مشاهده نمود:

```
public void assignNocontrol(BufferedReader reader)
throws IOException
{
	String line;
	while ((line = reader.readLine()) != null)
	{
		// ... Work with line
	}
}
```

۵,۵ از آکولاد برای عبارات بدنه‌ی *if*، *for*، و *while* استفاده کنید

از آکولادهای باز و بسته برای عبارات *if*، *for* و *while* استفاده کنید، حتی هنگامی‌که بدنه، شامل تنها یک عبارت ساده است. آکولاد، شکل و قابلیت خواندن کد را بهبود می‌بخشد. مهم‌تر از همه، اضافه نمودن آکولاد هنگام درج عبارات اضافه به بدنه‌ای که شامل یک عبارت ساده است، به آسانی فراموش می‌شود، زیرا دندانه‌دندانه نمودن[۱۵۱] مرسوم برنامه، هدایتی قوی (اما گمراه‌کننده) را برای ساختار ایجاد می‌کند.

۱,۵,۵ نمونه‌کد ناسازگار

این مثال، کاربری را توسط یک عبارت *if*، که فاقد آکولاد است، تایید می‌کند.

```
int login;
if (invalid_login())
	login = 0;
else
	login = 1;
```

این برنامه، همان‌گونه که انتظار می‌رود رفتار می‌کند. بااین‌حال، ممکن است بعدا یک نگهدارنده، یک وضعیت اشکال‌زدایی یا تصحیح خطایی را، که فاقد آکولاد باز و بسته است، اضافه نماید.

```
int login;
if (invalid_login())
	login = 0;
else
	// Debug line added below
	System.out.println("Login is valid\n");
	// The next line is always executed
	login = 1;
```

دندانه‌دندانه کردن کد، عملکرد و قابلیت برنامه را تغییر می‌دهد به‌صورت بالقوه، منجر به آسیب‌پذیری امنیتی می‌شود.

[۱۵۱] Indentation

۲،۵،۵ راه‌حل سازگار

این راه‌حل، حتی در صورتی‌که بدنه‌های *if* و *else* متعلق به عبارت *if*، تک‌عبارت (عبارت تک‌خط) باشند، از آکولاد باز و بسته استفاده می‌کند:

```
int login;
if (invalid_login())
{
	login = 0;
}
else
{
	login = 1;
}
```

۳،۵،۵ نمونه‌کد ناسازگار

در این مثال، یک عبارت *if* داخل عبارت *if* دیگر قرار می‌گیرد، بدون آن‌که آکولادی در دوطرف بدنه‌ی *if* و *else* قرار گرفته‌باشد:

```
int privileges;
if (invalid_login())
	if (allow_guests())
		privileges = GUEST;
else
	privileges = ADMINISTRATOR;
```

ممکن است دندانه‌دندانه نمودن، برنامه‌نویس را به این باور برساند که به کاربران، تنها هنگامی‌که ورود آن‌ها معتبر است، دسترسی مدیریتی داده شده‌است. در واقع، دستور *else* جزو عبارت *if* داخلی محسوب شده‌است. در نتیجه، این نقص به کابران نامعتبر، اجازه‌ی دسترسی مدیریتی می‌دهد.

```
int privileges;
if (invalid_login())
	if (allow_guests())
		privileges = GUEST;
	else
		privileges = ADMINISTRATOR;
```

۴،۵،۵ راه‌حل سازگار

این راه‌حل، از آکولاد برای ازبین بردن ابهام استفاده می‌کند. در نتیجه، این اطمینان را حاصل می‌نماید که دسترسی سطوح به‌درستی اعطا شده‌است:

```
int privileges;
if (invalid_login())
{
	if (allow_guests())
	{
		privileges = GUEST;
	}
}
else
{
	privileges = ADMINISTRATOR;
}
```

۵،۵،۵ کاربرد

عدم وجود آکولاد در عبارات *if*، *for* و *while* باعث می‌شود که کد در معرض خطا قرار گیرد و هزینه‌ی نگهداری افزایش یابد.

۶٫۵ یک نقطه‌ویرگول (;) را بلافاصله پس از عبارات شرطی *if*، *for*، و *while* به کار نبرید

یک نقطه‌ویرگول را بلافاصله پس از عبارات شرطی *if*، *for* و *while* به کار نبرید، زیرا عموما نشان‌دهنده‌ی خطای کد برنامه‌نویس است و منجر به رفتار غیرمنتظره می‌شود.

۱٫۶٫۵ نمونه‌کد ناسازگار

در این مثال، یک نقطه‌ویرگول، پس از شرط *if* استفاده می‌شود:

```
if (a == b);
{
	/* ... */
}
```

همواره، عبارات موجود در بدنه‌ی ظاهری عبارت *if*، علی‌رغم نتیجه‌ی عبارت شرطی، مورد ارزیابی قرار می‌گیرند.

۲٫۶٫۵ راه‌حل سازگار

این راه‌حل، نقطه‌ویرگول را حذف کرده و تضمین می‌کند که بدنه‌ی عبارت *if*، تنها هنگامی‌که عبارت شرطی درست باشد، اجرا می‌گردد.

```
if (a == b)
{
	/* ... */
}
```

۳٫۶٫۵ کاربرد

قراردادن یک نقطه‌ویرگول، بلافاصله پس از عبارات شرطی *if*، *for* و *while*، منجر به رفتار غیرمنتظره‌ای می‌شود.

۷٫۵ هرمجموعه‌ای از وضعیت‌های مرتبط با برچسب *case* را با یک وضعیت *break*، پایان دهید

یک بلوک *switch*، شامل چند برچسب *case* و یک برچسب *default* (به‌شدت پیشنهادی اما اختیاری) است. عبارات پیرو هر برچسب *case* باید با یک وضعیت *break*، که مسئول انتقال کنترل به پایان بلوک *switch* است، خاتمه یابند. در صورت نادیده‌گرفتن و حذف، عبارات برچسب *case* بعدی اجرا می‌شوند. به‌دلیل این‌که وضعیت *break* اختیاری است، حذف آن هیچ خطری را متوجه کامپایلر نخواهد کرد. هنگامی‌که این رفتار غیرعادی است، می‌تواند جریان کنترل غیرمنتظره‌ای را ایجاد نماید.

۱٫۷٫۵ نمونه‌کد ناسازگار

در این مثال، *case* مربوط به *card* حاوی مقدار 11، فاقد عبارت *break* است. در نتیجه، اجرا، با عبارت *card = 12* ادامه می‌یابد.

```
int card = 11;
switch (card)
{
	/* ... */
	case 11:
		System.out.println("Jack");
	case 12:
		System.out.println("Queen");
	break;
	case 13:
		System.out.println("King");
	break;
	default:
		System.out.println("Invalid Card");
	break;
}
```

۲٫۷٫۵ راه‌حل سازگار

دراین راه‌حل، هر *case*، با یک وضعیت *break* پایان می‌یابد.

```
int card = 11;
switch (card)
{
	/* ... */
	case 11:
		System.out.println("Jack");
	break;
	case 12:
		System.out.println("Queen");
	break;
	case 13:
		System.out.println("King");
	break;
	default:
		System.out.println("Invalid Card");
	break;
}
```

۳.۷.۵ کاربرد

عدم استفاده از عبارت *break*، منجر به جریان کنترلی غیرمنتظره‌ای می‌گردد. عبارت *break* موجود در پایان *case* می‌تواند حذف شود. براساس قرارداد، این یک برچسب *default* است. عبارت *break*، کنترل را به پایان بلوک *switch* منتقل می‌کند. این رفتار، منجر به حرکت نزولی کنترل به پایان بلوک *switch* می‌شود. در نتیجه، بدون درنظرگرفتن وجود یا عدم وجود وضعیت *break*، کنترل به وضعیت‌های موجود در بلوک *switch* منتقل می‌شود. با این‌وجود، با توجه به یک سبک برنامه‌نویسی صحیح، *case* نهایی باید از طریق یک عبارت *break*، پایان یابد.

استثنائا، هنگامی‌که نیاز باشد *case*های چندگانه به‌صورت یکسان اجرا شوند، عبارات *break* می‌توانند از تمامی *case*ها، به‌جز آخرین مورد، حذف شود. به‌طور مشابه، عبارت *break* می‌تواند هنگام پردازش یک *case*، که پیش‌پردازش مناسبی برای یک یا چند *case* دیگر به‌حساب می‌آید، از *case* پیشوندی حذف شود. این موضوع باید مشخصا با یک مثال توضیح داده‌شود:

```
int card = 11;
int value;
// Cases 11,12,13 fall through to the same case
switch (card)
{
	// Processing for this case requires a prefix
	// of the actions for the following three
	case 10:
		do_something(card);
		// Intentional fall-through
		// These three cases are treated identically
	case 11: // Break not required
	case 12: // Break not required
	case 13:
		value = 10;
	break; // Break required
	default:
		// Handle error condition
}
```

همچنین، هنگامی‌که یک *case* با یک عبارت *return* یا *throw* و یا یک متد غیربازگشتی مانند *System.exit()* پایان می‌یابد، عبارت *break* می‌تواند حذف شود.

۸,۵ از حرکت غیرعمدی شمارنده‌های حلقه اجتناب کنید

در صورت عدم کدنویسی درست و مناسب، حلقه‌های *while* یا *for* می‌توانند تا بی‌نهایت و یا تا زمانی‌که شمارنده به مقدار نهایی خود برسد، اجرا شوند. این مشکل ممکن است ناشی از افزایش یا کاهش یک‌واحدی یک شمارنده‌ی حلقه و سپس، بررسی برابری با یک مقدار مشخص برای اتمام حلقه، باشد. دراین‌مورد، ممکن است که شمارنده‌ی حلقه، از مقدار مشخص‌شده عبور کند و تا بی‌نهایت اجرا شود و یا تا زمانی‌که به مقدار نهایی خود برسد، اجرا گردد. همچنین، ممکن است این مشکل ناشی از تست نفوذ در برابر محدودیت‌ها باشد: برای مثال، تکرار حلقه در زمانی‌که شمارنده، کوچکتر-مساوی *Integer.MAX_VALUE* یا بزرگتر-مساوی *Integer.MIN_VALUE* است.

۱,۸,۵ نمونه‌کد ناسازگار

به‌نظر می‌رسد که این کد باید پنج بار تکرار شود.

```
for (i = 1; i != 10; i += 2)
{
      // ...
}
```

این در حالی است که حلقه هیچ‌گاه پایان نمی‌پذیرد. مقادیر مربوط به *i* عبارتند از ۱، ۳، ۵، ۷، ۹، ۱۱ و غیره. در واقع، هرگز مقدار *i* برابر با ۱۰ نخواهد شد. این مقدار، به ماکزیمم عدد مثبت قابل نمایش می‌رسد (*Integer.MAX_VALUE*). سپس، به‌سمت دومین کوچکترین عدد منفی (Integer.MIN_VALUE + 1) پیش می‌رود. پس از آن، راه خود را تا ۱- و ۱ ادامه می‌دهد و همان‌طور که قبلا شرح داده‌شد، ادامه می‌یابد.

۲,۸,۵ نمونه‌کد ناسازگار

در این مثال، حلقه پایان می‌یابد اما تعداد تکرار بیشتری، نسبت به آن‌چه که انتظار می‌رود، اجرا می‌شود:

```
for (i = 1; i != 10; i += 5)
{
      // ...
}
```

مقادیر مرتبط با *i* عبارتند از ۱، ۶، ۱۱ و غیره، که از ۱۰ پرش می‌کند. سپس، مقدار *i*، از جوار مقدار ماکزیمم مثبت به حوالی کمترین مقدار منفی و پس از آن، به سمت صفر حرکت می‌کند. سپس اعداد ۲، ۷، ۱۲ را می‌گیرد و بار دیگر از ۱۰ پرش می‌کند. پس از این‌که سه بار دیگر از سمت بزرگترین عدد مثبت به سمت کمترین عدد منفی رفت، در نهایت، به ۰، ۵ و ۱۰ می‌رسد و حلقه پایان می‌پذیرد.

۳.۸.۵ راه‌حل سازگار

یک راه‌حل ساده این است که پیش از آن‌که شمارنده به‌صورت غیرعمد حرکت داده‌شود، اطمینان حاصل گردد که شرایط خاتمه‌ی حلقه فرارسیده است.

```
for (i = 1; i == 11; i += 2)
{
	// ...
}
```

این راه‌حل زمانی می‌تواند شکننده باشد که یک یا چند شرط موثر بر تکرار، تغییر داده شوند. یک راه‌حل بهتر، استفاده از عملگرهای مقایسه‌ای عددی (مانند <، =<، >، یا =>) برای پایان حلقه است.

```
for (i = 1; i <= 10; i += 2)
{
	// ...
}
```

این راه‌حل، می‌تواند در برخورد با تغییرات شرایط تکرار، قوی‌تر عمل کند. با این‌حال، این رویکرد هرگز نباید جایگزین توجه دقیق در خصوص تعداد تکرارهای موردنظر و واقعی شود.

۴.۸.۵ نمونه‌کد ناسازگار

یک عبارت حلقه، که بررسی می‌کند آیا شمارنده‌ی حلقه، کوچکتر-مساوی *Integer.MAX_VALUE* است یا بزرگتر-مساوی *Integer.MIN_VALUE*، هرگز به پایان نخواهد رسید، زیرا همیشه با *true* ارزیابی خواهد شد. برای مثال، حلقه‌ی زیر هرگز به پایان نمی‌رسد، زیرا *i*، هرگز بزرگتر از *Integer.MAX_VALUE* نخواهد شد.

```
for (i = 1; i <= Integer.MAX_VALUE; i++)
{
	// ...
}
```

۵,۸,۵ راه‌حل سازگار

دراین مثال، هنگامی‌که *i* برابر با *Integer.MAX_VALUE* شود، حلقه به پایان می‌رسد:

```
for (i = 1; i < Integer.MAX_VALUE; i++)
{
	// ...
}
```

اگر حلقه برای هر مقدار بزرگتر از صفر *i*، به‌عنوان تکرار درنظر گرفته شود و شامل *Integer.MAX_VALUE* باشد، می‌تواند به شکل زیر پیاده‌سازی گردد:

```
i = 0;
do
{
	i++
	// ...
} while (i != Integer.MAX_VALUE);
```

۶,۸,۵ نمونه‌کد ناسازگار

این مثال، شمارنده‌ی حلقه (*i*) را به صفر مقداردهی اولیه می‌کند و سپس، آن را در هر تکرار، دو واحد افزایش می‌دهد. انتظار می‌رود هنگامی‌که *i* بزرگتر از *Integer.MAX_VALUE – 1* است، حلقه به پایان برسد. در این مورد، حلقه به پایان نمی‌رسد، زیرا شمارنده، پیش از این‌که بزرگتر از *Integer.MAX_VALUE – 1* شود، حرکت می‌کند.

```
for (i = 0; i <= Integer.MAX_VALUE - 1; i += 2)
{
	// ...
}
```

۷.۸.۵ راه‌حل سازگار

در این مثال، هنگامی‌که شمارنده‌ی *i* بزرگتر از *Integer.MAX_VALUE* منهای مقدار گامی شود که به‌عنوان شرایط پایان حلقه درنظر گرفته شده‌است، حلقه پایان می‌پذیرد.

```
for (i = 0; i <= Integer.MAX_VALUE - 2; i += 2)
{
      // ...
}
```

۸.۸.۵ کاربرد

پایان نادرست حلقه، منجر به حلقه‌ی بی‌نهایت، کارایی ضعیف، نتایج نادرست، و سایر مشکلات می‌گردد. اگر هرکدام از شرایط پایان‌پذیری حلقه بتوانند توسط یک مهاجم دستکاری شوند، این خطاها می‌تواند برای حملات انکار خدمات یا دیگر حملات، مورد بهره‌برداری قرار گیرند.

۹٫۵ برای اولویت‌دهی به اعمال، از پرانتز استفاده کنید

برنامه‌نویسان اغلب دچار خطا در استفاده از اولویت عملگرها می‌شوند، زیرا درک پایینی از سطوح اولویت &، |، ^، <<، و >> دارند. برای اجتناب از این اشتباه، از پرانتز استفاده کنید که باعث خوانایی بهتر کد نیز می‌شود.

۱٫۹٫۵ نمونه‌کد ناسازگار

در این مثال، هدف عبارت، اضافه کردن متغیر *OFFSET* به نتیجه‌ی *AND* منطقی سطح بیت بین *x* و *MASK* است.

```
public static final int MASK = 1337;
public static final int OFFSET = -1337;
public static int computeCode(int x)
{
	return x & MASK + OFFSET;
}
```

طبق دستورالعمل اولویت عملگر، عبارت به‌صورت زیر، تجزیه می‌شود:

```
x & (MASK + OFFSET)
```

این عبارت، به‌صورت زیر ارزیابی می‌شود و مقدار ۰ را نتیجه خواهد داد:

```
x & (1337 - 1337)
```

۲٫۹٫۵ راه‌حل سازگار

این راه‌حل، از پرانتز استفاده می‌کند تا اطمینان یابد که عبارت، همان‌گونه که مدنظر است، ارزیابی می‌شود:

```
public static final int MASK = 1337;
public static final int OFFSET = -1337;
public static int computeCode(int x)
{
	return (x & MASK) + OFFSET;
}
```

۳.۹.۵ نمونه‌کد ناسازگار

هدف این مثال، الحاق "0" یا "1" به رشته‌ی "*value=*" است.

```
public class PrintValue
{
	public static void main(String[] args)
	{
		String s = null;
		// Prints "1"
		System.out.println("value=" + s == null ? 0 : 1);
	}
}
```

با این‌حال، قوانین اولویت منجر می‌شوند که عبارت، به‌صورت *("value=" + s) == null ? 0 : 1* چاپ شود.

۴.۹.۵ راه‌حل سازگار

این راه‌حل، از پرانتز استفاده می‌کند تا اطمینان یابد که عبارت، همان‌گونه که مدنظر است، ارزیابی می‌شود:

```
public class PrintValue
{
	public static void main(String[] args)
	{
		String s = null;
		// Prints "value=0" as expected
		System.out.println("value=" + (s == null ? 0 : 1));
	}
}
```

۵.۹.۵ کاربرد

اشتباهات مربوط به قوانین اولویت می‌توانند منجر به ارزیابی نادرست یک عبارت گردند و در نتیجه، منجر به رفتار ناهنجار و غیرمنتظره برنامه شوند. ممکن است پرانتزها، از عبارات ریاضی که از قواعد اولویت جبری پیروی می‌کنند، حذف شوند. برای مثال، عبارت زیر را درنظر بگیرید:

```
x + y * z
```

براساس قرارداد ریاضی، ضرب، قبل از عمل جمع اجرا می‌شود. در نتیجه، پرانتزها در این مورد، اضافی هستند:

```
x + (y * z)
```

تشخیص تمام عبارات استفاده‌کننده از عملگرهای با اولویت کم، بدون پرانتزها، ساده است. تعیین صحت و درستی این‌گونه کاربردها، در موارد کلی غیرممکن است. اگرچه، هشدارهای هیوریستیک می‌تواند کاربردی و مفید باشد.

۱۰٫۵ هیچ فرضی مبنی بر ساخت و ایجاد فایل، ایجاد نکنید

اگرچه، ساخت یک فایل با فراخوانی یک متد ساده انجام می‌پذیرد، اما این عمل ساده، چندین سوال امنیتی را با خود به‌همراه دارد. اگر فایل نتواند ایجاد شود چه باید کرد؟ اگر فایل قبلا وجود داشته‌باشد چه باید کرد؟ ویژگی‌های اولیه‌ی فایل، مانند مجوزها، چه باید باشد؟

جاوا، چندین نسل از امکانات مدیریت فایل را فراهم می کند: امکانات ورودی/ خروجی اصلی، که شامل مدیریت فایل پایه است، در بسته‌ی *java.io* قرار دارد؛ امکانات جامع‌تر در *JDK 1.4*، در بسته‌ی جدید ورودی/خروجی *java.nio* قرار دارد؛ امکانات جامع‌تر جدید در *JDK 1.7*، در بسته‌ی جدید دو ورودی/خروجی *java.nio.file* قرار دارد. هردوی این بسته‌ها، تعدادی متد را برای کنترل دقیق روی ساخت فایل، معرفی می‌کنند.

۱٫۱۰٫۵ نمونه‌کد ناسازگار

این مثال، سعی در بازکردن فایلی برای نوشتن دارد:

```
public void createFile(String filename)
throws FileNotFoundException
{
	OutputStream out = new FileOutputStream(filename);
	// Work with file
}
```

اگر فایل قبل از بازکردن وجود داشته‌باشد، محتویات سابق آن با محتویات ارائه‌شده توسط برنامه، بازنویسی خواهدشد.

۲٫۱۰٫۵ نمونه‌کد ناسازگار (*TOCTOU*)

این مثال، تلاش دارد از طریق ساخت فایل خالی با استفاده از *java.io.File.createNewFile()*، از تغییر یک فایل موجود، جلوگیری به‌عمل آورد. اگر یک فایل با نام داده‌شده وجود داشته‌باشد، *createNewFile()*، مقدار *false* را بدون محو کردن محتویات فایل موردنظر، برخواهد گرداند.

```
public void createFile(String filename)
throws FileNotFoundException
{
	OutputStream out = new FileOutputStream(filename, true);
	if (!new File(filename).createNewFile())
	{
		// File cannot be created...handle error
	} else
	{
		out = new FileOutputStream(filename);
		// Work with file
	}
}
```

متاسفانه این راه‌حل، به شرایط مسابقه‌ی TOCTOU[۱۵۲] بستگی دارد. ممکن است یک مهاجم سیستم، فایل را پس از ایجاد فایل خالی اما پیش از باز کردن فایل، تغییر دهد، به‌طوری‌که فایل بازشده، متفاوت از فایل ایجادشده باشد.

۳.۱۰.۵ راه‌حل سازگار (*Files*)

این راه‌حل، از متد *java.nio.file.Files.newOutputStream()* برای ساخت ذره‌ای فایل استفاده می‌کند و اگر فایل از قبل وجود داشته‌باشد، یک استثنا را ایجاد می‌نماید.

```
public void createFile(String filename)
throws FileNotFoundException
{
	try (OutputStream out=new BufferedOutputStream(Files.newOutputStream
	(Paths.get(filename),StandardOpenOption.CREATE_NEW)))
	{
			// Work with out
	}
	catch (IOException x)
	{
		// File not writable...Handle error
	}
}
```

[۱۵۲] Time-Of-Check, Time-Of-Use (TOCTOU)

۴.۱۰.۵ کاربرد

توانایی تعیین باز شدن یک فایل موجود و یا ایجاد شدن یک فایل جدید، اطمینان بیشتری را نسبت به حالتی که فقط فایل موردنظر باز یا رونویسی شود و فایل‌های دیگر بدون اثر باقی بمانند، فراهم می‌آورد.

۱۱,۵ برای عملیات‌های اعشاری، اعداد صحیح را به اعشاری تبدیل کنید

استفاده‌ی بی‌ملاحظه از محاسبات عدد صحیح، به‌منظور محاسبه‌ی یک مقدار جهت تخصیص به یک متغیر اعشاری می‌تواند منجر به ازدست‌رفتن اطلاعات گردد. به‌عموان مثال همواره، محاسبات صحیح، نتایج صحیح تولید می‌نماید و اطلاعات باقی‌مانده‌ی کسری را حذف می‌کند. علاوه براین، هنگام تبدیل مقادیر صحیح به اعشاری، دقت می‌تواند نادیده گرفته شود. در برنامه‌نویسی صحیح، عباراتی که مقادیر صحیح و اعشاری را ترکیب می‌کنند، به بررسی دقیق نیاز دارند.

لازم است عملیاتی متأثر از سرریز عدد صحیح یا ازدست رفتن باقی‌مانده‌ی کسری، روی مقادیر اعشاری اجراشوند.

۱,۱۱,۵ نمونه‌کد ناسازگار

دراین نمونه‌کد، تقسیم و ضرب روی مقادیر صحیح انجام می‌شود. سپس، نتیجه‌ی این اعمال به اعشار تبدیل می‌گردد.

```
short a = 533;
int b = 6789;
long c = 4664382371590123456L;
float d = a / 7; // d is 76.0 (truncated)
double e = b / 30; // e is 226.0 (truncated)
double f = c * 2; // f is -9.1179793305293046E18 because of integer overflow
```

نتایج عملیات صحیح، به نزدیک‌ترین عدد صحیح گرد شده و همچنین می‌تواند سرریز شود. در نتیجه، متغیرهای اعشاری *d*، *e* و *f* به‌درستی مقداردهی اولیه نمی‌شوند، زیرا گرد کردن و سرریز، پیش از تبدیل به نوع اعشاری رخ می‌دهد. توجه داشته‌باشید، محاسبه‌ی c نیز راهنمای "NUM00-J. تشخیص یا جلوگیری از سرریز عدد صحیح" را نقض می کند.

۲,۱۱,۵ راه‌حل سازگار (اعشار شناخته‌شده)

این راه‌حل، اعمال ضرب و تقسیم را روی مقادیر اعشاری اجرا کرده و از گرد کردن و سرریز موجود در نمونه‌کد ناسازگار، اجتناب می‌کند. در هر عملیات، حداقل یک عملوند از نوع اعشار وجود دارد. اعمال ضرب و تقسیم، در قالب اعشاری صورت می‌پذیرد و از سرریز و گرد کردن، اجتناب به‌عمل می‌آید.

```
short a = 533;
int b = 6789;
long c = 4664382371590123456L;
float d = a / 7.0f; // d is 76.14286
double e = b / 30.; // e is 226.3
double f = (double)c * 2; // f is 9.328764743180247E18
```

راه‌حل دیگر، حذف خطاهای سرریز و گردکردن از طریق ذخیره‌ی اعداد صحیح در متغیرهای اعشاری، پیش از انجام عملیات محاسباتی است.

```
short a = 533;
int b = 6789;
long c = 4664382371590123456L;
float d = a;
double e = b;
double f = c;
d /= 7; // d is 76.14286
e /= 30; // e is 226.3
f *= 2; // f is 9.328764743180247E18
```

همانند راه‌حل پیشین، این راه‌حل نیز اطمینان حاصل می‌نماید که حداقل یکی از عملوندهای هر عمل، عددی اعشاری است.

درهردو راه‌حل، مقدار اصلی *c* نمی‌تواند دقیقا به‌عنوان یک *double* نمایش داده‌شود. نمایش نوع *double*، تنها دارای ۴۸ بیت مانتیس (جزء کسری) است، اما نمایش دقیق *c*، به ۵۶ بیت مانتیس نیاز دارد. در نتیجه، مقدار *c* به نزدیک‌ترین مقدار قابل نمایش توسط نوع *double* گرد می‌شود و مقدار محاسبه‌شده‌ی *f* (9.328764743180247E18)، متفاوت از نتیجه‌ی دقیق ریاضی (9328564743180246912) خواهد بود. این از دست رفتن دقت، یکی از دلایل متعدد برنامه‌نویسی صحیح عباراتی است که اعمال اعشاری و صحیح و یا مقادیر نیازمند توجه دقیق‌تر را با یکدیگر ترکیب می‌کند. با وجود این ازدست‌رفتن دقت، مقدار محاسبه‌شده‌ی *f*، دقیق‌تر از مقدار تولیدشده در نمونه‌کد ناسازگار است.

۳.۱۱.۵ نمونه‌کد ناسازگار

این مثال، تلاش دارد تمام اعداد بزرگتر از نسبت دو عدد صحیح را محاسبه نماید. انتظار می‌رود که نتیجه، ۲٫۰ باشد، اما ۱٫۰ است.

```
int a = 60070;
int b = 57750;
double value = Math.ceil(a/b);
```

به‌عنوان نتیجه‌ای از قوانین ارتقای عددی جاوا، عملیات تقسیم اجراشده، تقسیمی از نوع صحیح است که نتیجه‌ی آن به ۱ گرد می‌شود. سپس این نتیجه، پیش از آن‌که به متد *Math.ceil()* گذر کند، به *double* ارتقا می‌یابد.

۴,۱۱,۵راه‌حل سازگار

این راه‌حل، پیش از اجرای تقسیم، یکی از عملوندها را به‌عنوان *double* معرفی می‌کند.

```
int a = 60070;
int b = 57750;
double value = Math.ceil(a/((double) b));
```

در نتیجه، عملوند دیگر به‌صورت خودکار، به‌عنوان *double* معرفی می‌شود. عمل تقسیم، تقسیمی از نوع *double* است و مقدار درست ۲,۰ به *value* تخصیص داده‌می‌شود. همانند راه‌حل قبل، این راه‌حل نیز اطمینان حاصل می‌نماید که حداقل یکی از عملوندهای هر عمل، عددی اعشاری است.

۵,۱۱,۵کاربرد

تبدیل‌های نامناسب بین عدد صحیح و مقادیر اعشاری می‌تواند نتایج غیرمنتظره‌ای را، به‌خصوص مانند از دست دادن دقت، به‌بار آورد. در برخی موارد، این نتایج غیرمنتظره می‌تواند شامل سرریز یا سایر شرایط استثنایی باشد.

قابل قبول است که عملیات را پیش از تبدیل عدد صحیح به اعشاری و با استفاده از ترکیبی از مقادیر این دو نوع عدد انجام دهیم. برای مثال، استفاده از محاسبات عدد صحیح، نیاز به استفاده از متد *floor ()* را حذف می‌کند. هر نوع کدی از این قبیل، باید به‌وضوح مستند شود تا به نگهدارندگان آینده را در درک این‌که چنین رفتاری عمدی است، یاری رساند.

۱۲٫۵ اطمینان حاصل نمایید که متد *clone()*، *super.clone()* را فراخوانی می‌کند

کلونینگ[۱۵۳] یک زیرکلاس از یک کلاس غیرنهایی، که یک متد *clone()* را که قادر نیست *super.clone()* را فراخوانی نماید، تعریف می‌کند، شیای از کلاسی اشتباه را تولید می‌نماید. شی بازگشتی باید با فراخوانی *super.clone* به‌دست آید. اگر یک کلاس و همه‌ی زیرکلاس‌های آن (به‌جز شی) از این قرارداد اطاعت کنند، آن‌گاه، رابطه‌ی *x.clone().getClass() == x.getClass()* برقرار خواهد بود.

۱٫۱۲٫۵ نمونه کد ناسازگار

دراین مثال، متد *clone()* در کلاس *Base*، موفق به فراخوانی *super.clone()* نمی‌شود:

```
class Base implements Cloneable
{
	public Object clone() throws CloneNotSupportedException {
		return new Base();
	}
	protected void doLogic() {
		System.out.println("Superclass doLogic");
	}
}
class Derived extends Base
{
	public Object clone() throws CloneNotSupportedException {
		return super.clone();
	}
	protected void doLogic() {
		System.out.println("Subclass doLogic");
	}
	public static void main(String[] args)
	{
		Derived dev = new Derived();
		try {
			Base devClone = (Base)dev.clone(); // Has type Base
			// instead of Derived
			devClone.doLogic(); // Prints "Superclass doLogic"
			// instead of "Subclass doLogic"
		}
		catch (CloneNotSupportedException e) { /* ... */ }
	}
}
```

[۱۵۳] Cloning

درنتیجه، شی *devClone*، به جای *Derived*، نوع *Base* را خواهد داشت و متد *doLogic()* به‌درستی اعمال نمی‌شود.

۲,۱۲,۵ راه‌حل سازگار

این راه‌حل، فراخوانی *super.clone()* را در کلاس *Base* متعلق به متد *clone()*، به‌درستی انجام می‌دهد.

```
class Base implements Cloneable
{
	public Object clone() throws CloneNotSupportedException
	{
		return super.clone();
	}
	protected void doLogic()
	{
		System.out.println("Superclass doLogic");
	}
}
class Derived extends Base
{
	public Object clone() throws CloneNotSupportedException
	{
		return super.clone();
	}
	protected void doLogic()
	{
		System.out.println("Subclass doLogic");
	}
	public static void main(String[] args)
	{
		Derived dev = new Derived();
		try {
			// Has type Derived, as expected
			Base devClone = (Base)dev.clone();
			devClone.doLogic(); // Prints "Subclass doLogic"
			// as expected
		}
		catch (CloneNotSupportedException e) { /* ... */ }
	}
}
```

۳,۱۲,۵ کاربرد

شکست در فراخوانی *super.clone()*، منجر به یک شی کلون‌شده از نوع غیرصحیح می‌شود.

۱۳,۵ از توضیحات به‌صورت پیوسته و به شکلی خوانا استفاده کنید

ترکیب استفاده از توضیحات سنتی یا بلوک (با شروع از /* و پایان یافتن به */) و نظرات پایان خط (از // تا انتهای خط) می‌تواند منجر به کد گمراه‌کننده‌ای شود که می‌تواند خطایی را به‌همراه داشته‌باشد.

۱,۱۳,۵ نمونه کد ناسازگار

این مثال، توضیحات ترکیب‌شده‌ای را نشان می‌دهد که قابل فهم نیستند.

```
// */                      /* Comment, not syntax error */
f = g/**//h;               /* Equivalent to f = g / h; */
/*//*/ l();                /* Equivalent to l(); */
m = n//**/o + p;           /* Equivalent to m = n + p; */
a = b //*divisor:*/c + d; /* Equivalent to a = b + d; */
```

۲,۱۳,۵ راه‌حل سازگار

به‌شکلی سازگار از توضیحات استفاده کنید.

```
// Nice simple comment
int i; // Counter
```

۳,۱۳,۵ نمونه کد ناسازگار

به‌صورت‌های نادرست دیگری از توضیحات استفاده می‌شود که باید از آنها اجتناب گردد. این مثال، از کاراکتر /* برای شروع توضیح استفاده می‌کند، اما از کاراکتر */ برای پایان استفاده نمی‌نماید. در نتیجه، فراخوانی به متد امنیت حیاتی اجرا نمی‌شود. بازرس بررسی این صفحه می‌تواند به اشتباه فرض کند که کد، اجرا شده‌است.

```
/* Comment with end comment marker unintentionally omitted
security_critical_method();
/* Some other comment */
```

استفاده از یک ویرایش‌گر برای نشانه‌گذاری گرامر یا قالب‌دهی به کد، جهت شناسایی مشکلاتی مانند از دست‌رفتن پایان توضیحات، می‌تواند به تشخیص ازقلم‌افتادگی‌ها و غفلت‌های تصادفی کمک کند. از

آن‌جاکه از دست‌رفتن پایان توضیحات، از جمله خطاهایی هستند که غالبا اتفاق می‌افتند و بیشتر به‌عنوان یک اشتباه درنظر گرفته‌می‌شوند، این روش برای ارائه‌ی توضیح در مورد کد، توصیه نمی‌شود.

۴,۱۳,۵راه‌حل سازگار

این راه‌حل، توصیه می‌کند که کد به‌عنوان "مرده" علامت‌گذاری شود. همچنین، از توانایی کامپایلر در راستای حذف کد غیرقابل دسترس (مرده) بهره می‌جوید. کد داخل بلوک *if* باید از نظر گرامری، صحیح باشد. اگر بعدا سایر بخش‌های برنامه به‌نحوی تغییر یابند که منجر به خطاهای گرامری شوند، کد اجرانشده باید به‌روزرسانی شوند تا مشکل را به‌درستی حل نمایند. سپس، درصورت نیاز در آینده، برنامه‌نویس فقط نیاز دارد که دستورات اطراف وضعیت *if* و توضیح *NOTREACHED* را حذف کند.

توضیح *NOTREACHED* می‌تواند به برخی از کامپایلرها و ابزراهای آنالیز ایستا بگوید که در مورد این کد غیرقابل دسترس، پیغامی ندهند. همچنین یک مستند را تنظیم می‌کند.

```
if (false)
{
	/* Use of critical security method no longer necessary, for now */
	/* NOTREACHED */
	security_critical_method();
	/* Some other comment */
}
```

۵,۱۳,۵کاربرد

سردرگمی در دستورات اجراشده و دستوراتی که اجرا نمی‌شوند، می‌تواند منجر به خطاهای برنامه‌نویسی و آسیب‌پذیری‌هایی، مانند انکار خدمات، پایان غیرعادی برنامه، و نقض یکپارچگی داده، شود. این مشکل، با استفاده از محیط‌های توسعه‌ی تعاملی (IDEها)[۱۵۴] و ویرایش‌گرهایی که از فونت، رنگ و سایر مکانیزم‌ها برای ایجاد میان بین توضیحات و کد استفاده می‌کنند، کاهش می‌یابد. با این‌وجود، مشکل می‌تواند همچنان توسط چاپگر، هنگام بازخوانی کد منبع چاپ‌شده، آشکار شود. توضیحات تودرتو و استفاده‌ی ناسازگار از توضیحات می‌تواند به‌وسیله‌ی ابزارهای آنالیز ایستای مناسب، کشف شده و تشخیص داده‌شوند.

[۱۵۴] Interactive Development Environments (IDEs)

۱۴,۵ کدها و مقادیر زائد و غیرضروری را تشخیص دهید و حذف کنید

کدها و مقادیر زائد و غیرضروری می‌توانند به شکل کد مرده- کدهایی که اثری ندارند و مقادیری که در منطق برنامه استفاده نمی‌شوند- رخ دهند. کدهایی که هرگز اجرا نمی‌شوند به‌عنوان کد مرده شناخته می‌شوند. به‌طور کلی، حضور کدهای مرده نشان می‌دهد که یک خطای منطقی، در نتیجه‌ی تغییری در برنامه یا محیط برنامه رخ داده‌است. کد مرده اغلب یک برنامه را در طول کامپایل از حالت بهینه خارج می‌کند. با این‌حال، برای بهبود خوانایی و اطمینان از حل خطاهای منطقی، کد مرده باید شناسایی، درک و حذف شود.

کدهایی که اجرا می‌شوند اما هیچ عملیاتی را انجام نمی‌دهند، و یا کدهایی که یک تاثیر بی‌هدف دارند، بیشتر نتیجه‌ی یک خطای کد هستند و می‌توانند منجر به رفتار غیرمنتظره شوند. وضعیت‌ها و عباراتی که اثری ندارند باید شناسایی شده و از کد حذف شوند. بسیاری از کامپایلرهای مدرن می‌توانند در مورد کدهای بی‌اثر هشدار دهند.

حضور مقادیر بلااستفاده در کد می‌تواند نشان‌دهنده‌ی خطای منطقی قابل‌توجهی باشد. برای پیشگیری از این خطاها، مقادیر بلااستفاده باید شناسایی و از کد حذف شوند.

۱,۱۴,۵ نمونه‌کد ناسازگار (کد مرده)

این نمونه‌کد نشان می‌دهد که چگونه یک کدمرده می‌تواند به یک برنامه معرفی شود:

```
public int func(boolean condition) {
	int x = 0;
	if (condition) {
		x = foo();
		/* Process x */
		return x;
	}
	/* ... */
	if (x != 0) {
		/* This code is never executed */
	}
	return x;
}
```

شرط مربوط به دومین وضعیت *if*، (*x != 0*)، هرگز تحقق پیدا نخواهد کرد، زیرا تنها مسیری که *x* می‌تواند به آن تخصیص داده‌شود، یک مقدار غیر صفر است که با یک وضعیت *return* پایان می‌پذیرد.

۲,۱۴,۵راه‌حل سازگار

برای اصلاح کدمرده، برنامه‌نویس نه‌تنها باید تعیین کند که کد هرگز اجرا نشده‌است، بلکه باید تعیین نماید که آیا الزامی برای اجرای کد وجود داشته‌است یا خیر. سپس، این وضعیت را به‌درستی حل کند. این راه‌حل سازنده، پیشنهاد می‌نماید که کد مرده باید اجرا شود و بدین‌ترتیب، بدنه‌ی اولین عبارت شرطی، دیگر با بازگشت به پایان نمی‌رسد.

```
public int func(boolean condition) {
	int x = 0;
	if (condition) {
		x = foo();
		/* Process x */
	}
	/* ... */
	if (x != 0) {
		/* This code is now executed */
	}
	return 0;
}
```

۳,۱۴,۵نمونه‌کد ناسازگار (کد مرده)

در این مثال، تابع *length()* برای محدودکردن تعداد دفعات تکرار تابع *string_loop()* استفاده می‌شود. هنگامی‌که اشاره‌گر جاری، طول *str* است، شرط مربوط به وضعیت *if* در حلقه، محقق می‌شود. با این‌حال، به‌دلیل این‌که *i* همواره کمتر از *str.length()* است، پس هرگز رخ نخواهد داد.

```
public int string_loop(String str) {
	for (int i=0; i < str.length(); i++) {
		/* ... */
		if (i == str.length()) {
			/* This code is never executed */
		}
	}
	return 0;
}
```

۴,۱۴,۵راه‌حل سازگار

اصلاح مناسب کد مرده، به هدف برنامه‌نویس بستگی دارد. فرض بر این است که هدف، انجام عملیاتی مخصوص توسط آخرین کاراکتر در *str* است. بنابراین، وضعیت شرط برای بررسی این‌که آیا *i* به اشاره‌گر آخرین کاراکتر در *str* رجوع می‌کند یا خیر، تنظیم می‌گردد.

```
public int string_loop(String str) {
	for (int i=0; i < str.length(); i++) {
		/* ... */
		if (i == str.length()-1) {
			/* This code is now executed */
		}
	}
	return 0;
}
```

۵,۱۴,۵نمونه‌کد ناسازگار

دراین مثال، مقایسه‌ی *s* با *t* هیچ اثری ندارد:

```
String s;
String t;
// ...
s.equals(t);
```

احتمالا این خطا، نتیجه‌ی هدف برنامه‌نویس برای انجام کاری از طریق مقایسه است، اما کد برای تکمیل آن با شکست مواجه می‌شود.

۶,۱۴,۵راه‌حل سازگار

دراین راه‌حل، نتیجه‌ی مقایسه چاپ می‌شود:

```
String s;
String t;
// ...
if (s.equals(t)) {
	System.out.println("Strings equal");
} else {
	System.out.println("Strings unequal");
}
```

۷.۱۴.۵نمونه‌کد ناسازگار (مقادیر بی‌مصرف)

در این مثال، *p2* به مقداری که *bar()* باز می‌گرداند تخصیص داده می‌شود، اما این مقدار هرگز استفاده نمی‌گردد:

```
int p1 = foo();
int p2 = bar();
if (baz())
{
	return p1;
}
else
{
	p2 = p1;
}
return p2;
```

۸.۱۴.۵راه‌حل سازگار

این مثال، بسته به هدف برنامه‌نویس، می‌تواند به چند شکل مختلف تصحیح گردد. دراین راه‌حل، *p2* غیرضروری است. فراخوانی به *bar()* و *baz()* نیز، درصورتی‌که هیچ اثر جانبی نداشته‌باشند، قابل حذف است.

```
int p1 = foo();
bar(); /* Removable if bar() lacks side effects */
baz(); /* Removable if baz() lacks side effects */
return p1;
```

۹.۱۴.۵کاربرد

حضور کدهای مرده می‌تواند نشان‌دهنده‌ی خطاهای منطقی باشد که منجر به رفتار برنامه‌ای بی‌هدف می‌شود. راه‌هایی که می‌توان کد مرده را به یک برنامه معرفی کرد و نیز تلاش برای حذف آن، می‌تواند پیچیده باشد. در نتیجه، حل کدمرده می‌تواند فرآیند عمیقی باشد که به آنالیز قابل‌توجهی نیاز دارد.

در وضعیت‌های استثنایی، می‌تواند نرم‌افزار را برای تغییرات آینده مقاوم کند. یک مثال، وجود یک *case* پیش‌فرض در وضعیت *switch* است؛ حتی با این‌وجود که همه‌ی برچسب‌های ممکن *switch*، مشخص شده‌اند.

همچنین مجاز است کد مرده را به‌طور موقت، که ممکن است بعدا موردنیاز باشد، حفظ کنید. چنین مواردی باید توسط توضیح مناسبی، نشان داده‌شود.

وجود کد بی‌اثر می‌تواند نشان‌دهنده‌ی خطای منطقی باشد و منجر به رفتار غیرمنتظره و آسیب‌پذیری شود. مقادیر بلااستفاده در کد، نشان‌دهنده‌ی خطاهای غیرمنطقی قابل‌توجه هستند. کدها و مقادیر بی‌اثر می‌توانند توسط آنالیز ایستای مناسب، تشخیص داده‌شوند.

۱۵,۵ برای تکمیل منطقی تلاش کنید

آسیب‌پذیری نرم‌افزار می‌تواند نتیجه‌ی عدم توجه برنامه‌نویس به همه‌ی وضعیت‌های داده‌ای ممکن باشد.

۱,۱۵,۵ نمونه کد ناسازگار (زنجیره‌ی *if*)

این مثال، در بررسی این‌که a، نه b است و نه c، شکست می‌خورد. این امر می‌تواند یک رفتار درست در این‌مورد باشد، اما اگر a به‌صورت غیرمنتظره، مقدار متفاوتی را درنظر بگیرد، عدم محاسبه‌ی تمام مقادیر a می‌تواند منجر به خطای منطقی شود.

```
if (a == b)
{
	/* ... */
}
else if (a == c)
{
	/* ... */
}
```

۲,۱۵,۵ راه‌حل سازگار (زنجیره‌ی *if*)

این راه‌حل، شرایط غیزمنتظره را به‌صراحت بررسی می‌کند و آن را به‌صورتی مناسب، حل و مدیریت می‌نماید:

```
if (a == b)
{
	/* ... */
}
else if (a == c)
{
	/* ... */
}
else
{
	/* Handle error condition */
}
```

۳,۱۵,۵نمونه‌کد ناسازگار (*switch*)

حتی اگر *x* در این نمونه‌کد، بیانگر یک بیت (۰ یا ۱) باشد، ممکن است برخی خطاهای قبلی، *x* را مقدار متفاوتی درنظر بگیرند. در چنین وضعیت ناسازگاری، تشخیص و مقابله‌ی سریع، یافتن خطا را آسان‌تر می‌سازد.

```
switch (x)
{
	case 0: foo(); break;
	case 1: bar(); break;
}
```

۴,۱۵,۵راه‌حل سازگار (*switch*)

این راه‌حل، برچسب *default* را برای مدیریت تمامی مقادیر ممکن از نوع *int* فراهم می‌سازد:

```
switch (x)
{
	case 0: foo(); break;
	case 1: bar(); break;
	default: /* Handle error */ break;
}
```

۵,۱۵,۵نمونه‌کد ناسازگار (*Zune 30*)

این نمونه‌کد، از کد C، که در دستگاه پخش رسانه‌ای *Zune 30* ظاهر و منجر به قفل شدن بسیاری از دستگاه‌های پخش در نیمه‌شب سی‌ام دسامبر ۲۰۰۸ شد، اقتباس شده‌است. این نمونه‌کد، دربردارنده‌ی منطق غیرکاملی است که منجر به یک حمله‌ی انکار خدمات شده‌است.

تابع اصلی *ConvertDays()* از نوع C، در روتین‌های ساعت بلادرنگ (RTC)[۱۵۵] برای MC13783 PMIC RTC، تعداد روزها را از اول ژانویه‌ی ۱۹۸۰ می‌گیرد و سال دقیق و نیز تعداد روزها را، از اول ژانویه‌ی سال دقیق، محاسبه می‌کند. نقص کد، زمانی رخ می‌دهد که تعداد روزها مقدار ۳۶۶ را بگیرند،

[۱۵۵] Real-Time Clock (RTC)

زیرا حلقه هرگز به پایان نمی‌رسد. این اشکال، خود را در ۳۶۶مین روز سال ۲۰۰۸ نمایان کرد، که اولین سال کبیسه از زمان فعالیت کد بود.

```
final static int ORIGIN_YEAR = 1980;
/* Number of days since January 1, 1980 */
public void convertDays(long days)
{
	int year = ORIGIN_YEAR;
	/* ... */
	while (days > 365)
	{
		if (IsLeapYear(year))
		{
			if (days > 366)
			{
				days -= 366;
				year += 1;
			}
		}
		else
		{
			days -= 365;
			year += 1;
		}
	}
}
```

۶,۱۵,۵راه‌حل سازگار (*Zune 30*)

تضمین می‌گردد که به‌دلیل کاهش *days* در هر تکرار از حلقه، حلقه خارج شود، مگر آن‌که شرط *while* با شکست مواجه شود، که در این‌صورت، حلقه خاتمه می‌یابد. این راه‌حل، برای اهداف نمایشی است و ممکن است متفاوت از راه‌حل پیاده‌سازی‌شده توسط مایکروسافت باشد.

```
final static int ORIGIN_YEAR = 1980;
/* Number of days since January 1, 1980 */
public void convertDays(long days)
{
	int year = ORIGIN_YEAR;
	/* ... */
	int daysThisYear = (IsLeapYear(year) ? 366 : 365);
	while (days > daysThisYear)
	{
		days -= daysThisYear;
		year += 1;
		daysThisYear = (IsLeapYear(year) ? 366 : 365);
	}
}
```

۷.۱۵.۵ کاربرد

عدم توجه به همه‌ی امکانات در یک وضعیت منطقی، می‌تواند منجر به وضعیت در حال تخریبی شود که می‌تواند به‌طور بالقوه، افشای اطلاعات ناخواسته یا پایان غیرعادی برنامه را به‌همراه داشته‌باشد.

۱۶٫۵ از سربار، به‌صورت مبهم یا گیج‌کننده، استفاده نکنید

سربار تابع‌سازنده و متد، امکان اعلان متدها و توابع سازنده را با همان نام اما با لیست‌های پارامتری متفاوت، فراهم می‌کند. کامپایلر، هرفراخوانی به یک متد یا تابع سازنده‌ی سربار را بازبینی می‌نماید و از انواع اعلانی پارامترهای متد، برای تصمیم این‌که کدام متد را فراخوانی کند، استفاده می‌نماید. با این‌حال دربرخی موارد، ممکن است با وجود ویژگی‌های زبانی نسبتا جدید، مانند باکسینگ خودکار و عمومیات، سردرگمی ایجاد شود.

علاوه‌براین، توابع سازنده و متدهای حاوی انواع پارامترهای یکسان و ترتیب اعلان متفاوت، توسط کامپایلر جاوا پرچم‌دار نمی‌شوند. خطاها هنگامی رخ می‌دهند که یک توسعه‌دهنده نتواند در هربار استفاده از تابع سازنده و متد، از مستند بهره‌برداری نماید. حوادث، مربوط به این است که معانی مختلفی، با هریک از متدها یا توابع سازنده سربار، در ارتباط است. برخی اوقات، تعریف معانی متفاوت به ترتیب متفاوتی از پارامترهای همان متد، برای ساخت یک چرخه‌ی نادرست و بداندیش نیاز دارد. برای مثال، یک متد *getDistance()* را در نظر بگیرید که در آن، متد، سربار فاصله پیموده‌شده از منبع را بازمی‌گرداند درحالی‌که متد دیگر (با پارامترهایی که مجددا مرتب شده‌اند)، فاصله‌ی باقیمانده تا مقصد را برمی‌گرداند. ممکن است پیاده‌سازی‌کنندگان، این تفاوت را درک نکنند، مگر این‌که در هر استفاده، به مستندات رجوع کنند.

۱٫۱۶٫۵ نمونه‌کد ناسازگار (سازنده)

توابع سازنده نمی‌توانند لغو شوند، بلکه تنها می‌توانند سربار گردند. این مثال، کلاس *Con* را با سه تابع سازنده‌ی سربار نشان می‌دهد:

```
class Con {
	public Con(int i, String s)
	{
		// Initialization Sequence #1
	}
	public Con(String s, int i)
	{
		// Initialization Sequence #2
	}
	public Con(Integer i, String s)
	{
		// Initialization Sequence #3
	}
}
```

عدم توجه و تجربه در هنگام ارسال پارامتر به این توابع سازنده می‌تواند سردرگمی ایجاد کند، زیرا فراخوانی به این توابع، شامل همان تعداد پارامترهای مشابه واقعی تایپ‌شده است. هنگامی‌که متدها یا توابع سازنده‌ی سربار، معانی مشخص و متمایزی را از پارامترهای رسمی انواع مشابه (که تنها تفاوت آنها در ترتیب اعلان است) فراهم می‌کنند، از سربار نیز باید جلوگیری به‌عمل آید.

۲,۱۶,۵ راه‌حل سازگار

این راه‌حل، از طریق اعلان متدهای کارخانه‌ای ایستای عمومی (که نام‌های متمایز و مشخصی را دارند) به‌جای توابع سازنده کلاس عمومی، از سربار جلوگیری می‌کند.

```
public static Con createCon1(int i, String s)
{
	/* Initialization Sequence #1 */
}
public static Con createCon2(String s, int i)
{
	/* Initialization Sequence #2 */
}
public static Con createCon3(Integer i, String s)
{
	/* Initialization Sequence #3 */
}
```

۳.۱۶.۵نمونه‌کد ناسازگار (متد)

در این مثال، کلاس *OverLoader*، یک نمونه *HashMap* را نگه می‌دارد و متد، دارای سربار *getData()* است. یک متد *getData()*، رکوردی را برای بازگشت براساس مقدار کلیدی خود در نقشه، انتخاب می‌کند. مابقی نیز براساس مقدار نگاشت‌شده‌ی واقعی، انتخاب می‌کنند.

```
class OverLoader extends HashMap<Integer,Integer> {
	HashMap<Integer,Integer> hm;
	public OverLoader() {
		hm = new HashMap<Integer, Integer>();
		// SSN records
		hm.put(1, 111990000);
		hm.put(2, 222990000);
		hm.put(3, 333990000);
	}
	public String getData(Integer i) { // Overloading sequence #1
		String s = get(i).toString(); // Get a particular record
		return (s.substring(0, 3) + "-" + s.substring(3, 5) + "-" +
			s.substring(5, 9));
	}
	public Integer getData(int i) { // Overloading sequence #2
		return hm.get(i); // Get record at position 'i'
	}
	// Checks whether the ssn exists
	@Override public Integer get(Object data) {
		// SecurityManagerCheck()
		for (Map.Entry<Integer, Integer> entry : hm.entrySet()) {
			if (entry.getValue().equals(data)) {
				return entry.getValue(); // Exists
			}
		}
		return null;
	}
	public static void main(String[] args) {
		OverLoader bo = new OverLoader();
		// Get record at index '3'
		System.out.println(bo.getData(3));
		// Get record containing data '111990000'
		System.out.println(bo.getData((Integer)111990000));
	}
}
```

برای اهداف تفکیک سربار، امضاهای متدهای *getData()* تنها در نوع ایستای پارامترهای رسمی خود تفاوت دارند. کلاس *OverLoader*، از *java.util.HashMap* ارث می‌برد و متد *get()* را برای تامین قابلیت بررسی، لغو می‌نماید. این پیاده‌سازی می‌تواند برای آن مشتری، که انتظار رفتار مشابهی از هردو متد *get()* را دارد، گیج‌کننده باشد و به این موضوع که آیا شاخص رکورد یا مقدار قابل بازیابی مشخص شده‌است یا خیر، بستگی ندارد.

اگرچه، ممکن است در نهایت، برنامه‌نویس سمت مشتری، نتیجه‌ی چنین رفتاری را استنباط کند و مابقی موارد، مانند رابط *List*، موردتوجه نباشد. در نتیجه، محتمل است که یک برنامه‌نویس متوجه حذف عنصر اشتباه از لیست نشود.

مشکل دیگر این است که در حضور باکسینگ خودکار، اضافه‌کردن تعریف جدید متد سربار می‌تواند کد قبلی درحال‌کار سمت مشتری را بشکند. این موضوع می‌تواند زمانی رخ دهد که یک متد سربار جدید با یک نوع خاص‌تر به یک API که متد آن، از انواع کمتر خاصی در نسخه‌های پیشین استفاده می‌کند، اضافه گردد. به‌عنوان مثال، اگر نسخه‌ی قبلی از کلاس سربار، تنها متد *getData(Integer)* را فراهم سازد، مشتری می‌تواند این متد را با گذر یک پارامتر از نوع *int*، به‌درستی فراخوانی کند. نتیجه می‌تواند بر پایه‌ی مقدار آن انتخاب شود، زیرا پارامتر *int* می‌تواند به‌صورت خودکار به *Integer* باکسینگ شود.

پس از آن، هنگامی‌که متد *getData(int)* اضافه شد، کامپایلر همه‌ی فراخوانی‌هایی را که پارامتر آنها از نوع *int* است، به‌منظور فراخوانی متد جدید *getData(int)*، تحلیل می‌کند. در نتیجه، معانی آنها را تغییر می‌دهد و به‌طور بالقوه، باعث شکستن کد صحیح قبلی می‌شود. کامپایلر در چنین مواردی، کاملا درست است؛ مشکل واقعی، یک تغییر ناسازگار با API است.

۴,۱۶,۵راه‌حل سازگار (متد)

نام‌گذاری متفاوت دو متد مرتبط، هر دوی سربار و سردرگمی را حذف می‌کند:

```
public Integer getDataByIndex(int i)
{
	// No longer overloaded
}
public String getDataByValue(Integer i)
{
	// No longer overloaded
}
```

۵,۱۶,۵ کاربرد

استفاده‌ی مبهم یا گیج‌کننده از سربار، منجر به نتیجه‌ی غیرمنتظره می‌شود.

فصل ششم: تصورات نادرست برنامه‌نویس

راهنماهای این بخش، به حوزه‌هایی اشاره می‌کنند که توسعه‌دهندگان، اغلب فرضیات غیرقابل ضمانتی را در آنها، در مورد زبان جاوا و رفتارهای کتابخانه و یا جایی که به‌راحتی می‌تواند ابهاماتی به‌وجود آید، تصور می‌کنند. شکست در پیروی از این راهنماها، می‌تواند منجر به برنامه‌هایی شود که نتایج خلاف واقعی[۱۵۶] را تولید می‌کنند. این بخش، حاوی راهنماهایی است که به موارد زیر می‌پردازند:

- تصورات نادرست در مورد APIهای جاوا و ویژگی‌های زبان
- فرضیات و برنامه‌های مبهم
- اوضاعی که در آن برنامه‌نویس قصد انجام کاری را دارد، اما در نهایت، کار دیگری را انجام می‌دهد.

[۱۵۶] Counterintuitive

1.6 فرض نکنید که اعلان یک ناپای مرجع، انتشار امن اعضای شی ارجاع داده‌شده را تضمین می‌نماید

یک فیلد می‌تواند به‌عنوان *volatile* اعلان شود. در این‌صورت، مدل حافظه‌ی جاوا اطمینان می‌یابد که تمام نخ‌ها، مقدار ثابتی[۱۵۷] را برای آن متغیر، مشاهده می‌نمایند. این تضمین انتشار امن، تنها به فیلدهای اصلی[۱۵۸] و ارجاعات شی، اعمال می‌شود.

معمولاً برنامه‌نویسان، از واژگان غیردقیق استفاده کرده و در مورد "اشیای عضو" صحبت می‌کنند. به‌منظور تضمین این رؤیت‌پذیری، عضو حقیقی، مرجع شی است. اشیایی که توسط ارجاعات شی *volatile* به آنها ارجاع داده می‌شود (یا مراجع[۱۵۹])، فراتر از حوزه‌ی این تضمین ایمنی هستند. در نتیجه، اعلان یک مرجع شی به‌عنوان *ناپا*[۱۶۰]، برای تضمین این‌که تغییرات در اعضای مراجع، به نخ‌های دیگر منتشر شوند، کافی نیست. ممکن است یک نخ، نتوان نوشتن جدید نخی دیگر را روی یک فیلد عضو چنین شی مرجعی، مشاهده نماید.

علاوه بر این، هنگامی‌که مرجع، تغییرپذیر و فاقد ایمنی نخ است، ممکن است یاسر نخ‌ها، یک شی جزئی ساخته‌شده یا یا یک شی در وضعیت ناسازگار (موقت) را ببینند. با این‌وجود، هنگامی‌که مرجع، غیرقابل تغییر است، اعلان ارجاع ناپا برای تضمین انتشار امن اعضای مرجع، کافی است. برنامه‌نویسان نمی‌توانند از کلمه‌ی کلیدی *volatile* برای تضمین انتشار امن اشیای تغییرپذیر استفاده کنند. استفاده از کلمه‌ی کلیدی *volatile*، تنها می‌تواند انتشار امن فیلدهای اصلی، ارجاعات شی، یا فیلدهای مربوط به مراجع شی تغییرناپذیر را تضمین نماید.

۱.۱.۶ نمونه‌کد ناسازگار (آرایه‌ها)

این نمونه‌کد، یک مرجع ناپا را برای یک شی آرایه‌ای، اعلان می‌کند:

---

[۱۵۷] Consistent
[۱۵۸] Primitive
[۱۵۹] Referents
[۱۶۰] Volatile

```
final class Foo
{
	private volatile int[] arr = new int[20];
	public int getFirst()
	{
		return arr[0];
	}
	public void setFirst(int n)
	{
		arr[0] = n;
	}
	// ...
}
```

ممکن است مقادیر نسبت داده‌شده به یک عنصر آرایه توسط یک نخ- مثلاً با فراخوانی *setFirst()*- برای نخ دیگری که *getFirst()* را فراخوانی می‌کند، قابل مشاهده نباشد، زیرا کلمه‌ی کلیدی *volatile*، انتشار امن را تنها برای ارجاع آرایه تضمین می‌نماید و هیچ تضمینی برای داده‌ی واقعی درون آرایه ندارد.

این مساله، زمانی به‌وجود می‌آید که نخ‌هایی که به‌ترتیب، *setFirst()* و *getFirst()* را فراخوانی می‌کنند، فاقد رابطه‌ی *"از پیش رخ‌داده"* باشند. این رابطه، بین نخی که در یک متغیر ناپا *می‌نویسد* و نخی که *متعاقباً آن را می‌خواند*، وجود دارد. با این‌حال، هریک از متدهای *setFirst()* و *getFirst()*، از یک متغیر ناپا می‌خوانند— ارجاع ناپا به آرایه. هیچ‌یک از این دو متد، در متغیر ناپا نمی‌نویسند.

۲,۱,۶ راه‌حل سازگار (*AtomicIntegerArray*)

به‌منظور حصول اطمینان از این‌که نوشتن‌ها در عناصر آرایه، تجزیه‌ناپذیر (اتمیک)[۱۶۱] بوده و مقادیر حاصل، برای نخ‌های دیگر، قابل مشاهده است، این راه‌حل سازگار، از کلاس *AtomicIntegerArray* تعریف‌شده در *java.util.concurrent.atomic* استفاده می‌کند:

[۱۶۱] Atomic

```
final class Foo
{
	private final AtomicIntegerArray atomicArray=new AtomicIntegerArray(20);
	public int getFirst()
	{
		return atomicArray.get(0);
	}
	public void setFirst(int n)
	{
		atomicArray.set(0, 10);
	}
	// ...
}
```

AtomicIntegerArray تضمین می‌کند که رابطه‌ی از پیش رخ‌داده‌ای بین نخی که *atomicArray.set()* را فراخوانی می‌کند و نخی که متعاقباً *atomicArray.get()* را فراخوانی خواهد کرد، وجود دارد.

۳.۱.۶ راه‌حل سازگار (همگام‌سازی)

به‌منظور حصول اطمینان حاصل کردن از رؤیت‌پذیری، متدهای تابع باید دسترسی را، صرف‌نظر از این‌که با یک ارجاع ناپا یا پایا، به آن آرایه اشاره شده‌باشد، در حین انجام عملیات روی عناصر پایای یک آرایه، همگام‌سازی کنند. توجه داشته‌باشید، اگرچه که ارجاع آرایه، پایا است، اما کد، از نوع ایمنی نخ است.

```
final class Foo
{
	private int[] arr = new int[20];
	public synchronized int getFirst()
	{
		return arr[0];
	}
	public synchronized void setFirst(int n)
	{
		arr[0] = n;
	}
}
```

همگام‌سازی، یک رابطه‌ی از پیش رخ‌داده را بین نخ‌هایی که روی یک قفل مشابه همگام‌سازی می‌شوند، ایجاد می‌نماید. در این مورد، نخی که *setFirst()* را فراخوانی می‌کند و نخی که متعاقباً *getFirst()* را روی نمونه شی مشابهی فراخوانی می‌نماید، هر دو، روی آن نمونه، همگام‌سازی می‌شوند.

۴.۱.۶ نمونه‌کد ناسازگار (شی تغییرپذیر)

این نمونه‌کد ناسازگار، فیلد نمونه‌ی *Map* را به‌صورت ناپا اعلان می‌کند. نمونه شی *Map*، به‌دلیل متد *put()* آن، تغییرپذیر است.

```
final class Foo
{
	private volatile Map<String, String> map;
	public Foo()
	{
		map = new HashMap<String, String>();
		// Load some useful values into map
	}
	public String get(String s)
	{
		return map.get(s);
	}
	public void put(String key, String value)
	{
		// Validate the values before inserting
		if (!value.matches("[\\w]*"))
		{
			throw new IllegalArgumentException();
		}
		map.put(key, value);
	}
}
```

ممکن است فراخوانی‌های متناوب[۱۶۲] *get()* و *put()*، منجر به بازیابی مقادیری شوند که به لحاظ داخلی، ناسازگار با شی *Map* هستند، زیرا *put()*، وضعیت خود را اصلاح می‌کند. اعلان ارجاع ناپای یک شی، برای حذف این رقابت داده‌ای کافی نیست.

[۱۶۲] Interleaved

۵,۱,۶ نمونه‌کد ناسازگار (خواندن ناپا، نوشتن همگام‌شده)

این نمونه‌کد، تلاش دارد از روش خواندن فَرار، نوشتن همگام‌شده[۱۶۳]، استفاده نماید. فیلد *map*، به‌صورت ناپا اعلان شده‌است تا خواندن و نوشتن‌های خود را همگام سازد. متد *put()* نیز، همگام‌سازی شده‌است تا از اجرای خودکار آن، اطمینان حاصل گردد.

```
final class Foo
{
	private volatile Map<String, String> map;
	public Foo()
	{
		map = new HashMap<String, String>();
		// Load some useful values into map
	}
	public String get(String s)
	{
		return map.get(s);
	}
	public synchronized void put(String key, String value)
	{
		// Validate the values before inserting
		if (!value.matches("[\\w]*"))
		{
			throw new IllegalArgumentException();
		}
		map.put(key, value);
	}
}
```

روش خواندن ناپا، نوشتن همگام‌شده، از همگام‌سازی برای حفظ تجزیه‌پذیری عملیات چندجزئی[۱۶۴]، مانند افزایش، استفاده می‌کند و زمان دسترسی سریع‌تری را برای خواندن‌های تجزیه‌پذیر، فراهم می‌آورد. با این‌وجود، در زمینه‌ی اشیای تغییرپذیر، با شکست مواجه می‌شود، زیرا تضمین انتشار امن مهیاشده توسط ناپایی، تنها به خود فیلد بسط پیدا می‌کند (مقدار اصلی یا ارجاع شی)؛ مرجع و اعضای آن، از تضمین، خارج می‌شوند. در عمل، یک نوشتن و پس از آن خواندن *map*، فاقد رابطه‌ی از پیش رخداده هستند.

---

[۱۶۳] Volatile-read, synchronized-write

[۱۶۴] Compound

۶,۱,۶ راه‌حل سازگار (همگام‌شده)

این راه‌حل، از همگام‌سازی متد برای تضمین رؤیت‌پذیری استفاده می‌کند:

```
final class Foo
{
	private final Map<String, String> map;
	public Foo()
	{
		map = new HashMap<String, String>();
		// Load some useful values into map
	}
	public synchronized String get(String s)
	{
		return map.get(s);
	}
	public synchronized void put(String key, String value)
	{
		// Validate the values before inserting
		if (!value.matches("[\\w]*"))
		{
			throw new IllegalArgumentException();
		}
		map.put(key, value);
	}
}
```

اعلان ناپای فیلد *map*، غیرضروری است، زیرا متدهای تابع همگام‌شده هستند. فیلد، به‌صورت *final* اعلان شده‌است تا هنگامی‌که مرجع تا حدی مقداردهی اولیه شده‌است، جلوی انتشار ارجاع آن را بگیرد (TSM03-J. اشیایی را که تا حدی مقداردهی اولیه شده‌اند، منتشر نکنید).

۷,۱,۶ نمونه‌کد ناسازگار (زیرشی تغییرپذیر)

در این نمونه‌کد، فیلد *format* ناپا، ارجاع به یک شی تغییرپذیر *java.text.DateFormat* را ذخیره می‌کند. به‌دلیل آن‌که *java.text.DateFormat* از نوع ایمنی نخ نیست، ممکن است مقدار *Data*، که توسط متد *parse()* برگردانده می‌شود، مرتبط با پارامتر *str* نباشد.

```
final class DateHandler
{
	private static volatile DateFormat format =
		DateFormat.getDateInstance(DateFormat.MEDIUM);
	public static java.util.Date parse(String str)
	throws ParseException
	{
		return format.parse(str);
	}
}
```

۸,۱,۶ راه‌حل سازگار (نمونه‌ای در هر فراخوانی / کپی تدافعی)

این راه‌حل، یک نمونه‌ی *DateFormat* را برای هر فراخوانی متد *parse()* می‌سازد و برمی‌گرداند:

```
final class DateHandler
{
	public static java.util.Date parse(String str)
	throws ParseException
	{
		return DateFormat.getDateInstance(DateFormat.MEDIUM).parse(str);
	}
}
```

۹,۱,۶ راه‌حل سازگار (همگام‌سازی)

این راه‌حل، *DateHandler* را با استفاده از همگام‌سازی عبارت‌های داخل متد *parse()*، از نوع ایمنی نخ می‌سازد:

```
final class DateHandler
{
	private static DateFormat format =
		DateFormat.getDateInstance(DateFormat.MEDIUM);
	public static java.util.Date parse(String str)
	throws ParseException
	{
		synchronized (format)
		{
			return format.parse(str);
		}
	}
}
```

۱۰,۱,۶ راه‌حل سازگار (ذخیره‌سازی *ThreadLocal*)

این راه‌حل، از یک شی *ThreadLocal* برای ساخت یک نمونه‌ی *DateFormat* مجزا برای هر نخ، استفاده می‌کند:

```
final class DateHandler
{
	private static final ThreadLocal<DateFormat> format =
		new ThreadLocal<DateFormat>()
		{
			@Override protected DateFormat initialValue()
			{
				return DateFormat.getDateInstance(DateFormat.MEDIUM);
			}
		};
	// ...
}
```

۱۱,۱,۶ کاربرد

این فرض اشتباه، که اعلان ناپای یک فیلد، انتشار امن اعضای یک شی مورد ارجاع را تضمین می‌کند، می‌تواند منجر به مشاهده‌ی مقادیر قدیمی یا ناسازگار توسط نخ شود.

از لحاظ فنی، تغییرناپذیری شدید یک مرجع، شرط قوی‌تری نسبت به آنچه که اساساً برای انتشار امن مورد نیاز است، محسوب می‌شود. وقتی بتوان تعیین نمود که مرجعی، توسط ایمنی نخ طراحی شده‌است، فیلدی که ارجاع آن را نگهداری می‌کند می‌تواند به‌صورت ناپا اعلان گردد. با این‌حال این روش، قابلیت نگهداری را به‌منظور استفاده از ناپایی، کاهش می‌دهد و باید از آن دوری کرد.

۲٫۶ تصور نکنید که متدهای *sleep()*، *yield()* یا *getState()*، سمانتیک‌های همگام‌سازی را فراهم می‌سازند

توجه داشته‌باشید، هر دوی *Thread.sleep* و *Thread.yield*، فاقد سمانتیک همگام‌سازی هستند. به‌طور مشخص، کامپایلر مجبور نیست نوشتن‌های ذخیره‌شده در حافظه‌ی نهان ثبّات‌ها را پیش از فراخوانی *Thread.sleep* یا *Thread.yield*، در حافظه‌ی مشترک ذخیره کند، یا این‌که لازم نیست کامپایلر مقادیر ذخیره‌شده در حافظه‌ی نهان ثبّات‌ها را پس از فراخوانی *Thread.sleep* یا *Thread.yield*، مجدداً بارگذاری نماید. کدی که ایمنی همروندی آن بر پایه‌ی تعلیق نخ استوار است یا منجر به فرایندهایی می‌شود که

- ثبّات‌های ذخیره‌شده در حافظه‌ی نهان را خالی می‌کنند،
- هر مقداری را مجددا بارگذاری می‌نمایند،
- هر رابطه‌ی از پیش رخداده‌ای را هنگام از سر گرفتن[۱۶۵] اجرا فراهم می‌سازند،

نادرست است و درنتیجه، مجاز نیست. برنامه‌ها باید اطمینان یابند که ارتباط بین نخ‌ها، همگام‌سازی، از پیش رخدادگی، و سمانتیک‌های انتشار امن را به‌طور مناسب داشته‌باشد.

۱٫۲٫۶ نمونه‌کد ناسازگار (*sleep()*)

این نمونه‌کد، تلاش دارد تا از عضو *Boolean* پایای *done*، به‌عنوان پرچمی برای پایان دادن به اجرای یک نخ استفاده کند. یک نخ مجزا، مقدار *done* را با فراخوانی متد *shutdown()*، *true* قرار می‌دهد.

کامپایلر در این مورد، آزاد است تا فیلد *this.done* را یک‌بار بخواند و در هر اجرای حلقه، مجدداً از مقدار ذخیره‌شده در حافظه‌ی نهان، استفاده کند. در نتیجه، ممکن است حلقه‌ی *while* هرگز پایان نیابد، حتی هنگامی‌که نخ دیگری، متد *shutdown()* را برای تغییر مقدار *this.done*، فراخوانی کند. این خطا می‌تواند ناشی از برنامه‌نویس باشد که به اشتباه فرض کرده‌بود فراخوانی *Thread.sleep()*، منجر به بارگذاری مجدد مقادیر ذخیره‌شده در حافظه‌ی نهان می‌شود.

---

[۱۶۵] Resumes

```
final class ControlledStop implements Runnable
{
	private boolean done = false;
	@Override public void run()
	{
		while (!done)
		{
			try
			{
				Thread.sleep(1000);
			}
			catch (InterruptedException e)
			{
				// Reset interrupted status
				Thread.currentThread().interrupt();
			}
		}
	}
	public void shutdown()
	{
		this.done = true;
	}
}
```

۲,۲,۶ راه‌حل سازگار (پرچم ناپا)

این راه‌حل، فیلد پرچم را، *volatile* اعلان می‌کند تا اطمینان یابد به‌روزرسانی‌های مقدار آن، برای چند نخ قابل مشاهده است:

```
final class ControlledStop implements Runnable
{
	private volatile boolean done = false;
	@Override public void run()
	{
		//...
	}
	// ...
}
```

کلمه‌ی کلیدی *volatile*، یک رابطه‌ی از پیش رخداده‌ای را میان این نخ و هر نخ دیگری که *done* را مقداردهی می‌کند، برقرار می‌نماید.

۳,۲,۶ راه‌حل سازگار (*Thread.interrupt()*)

یک راه‌حل بهتر برای متدهایی که *sleep()* را فراخوانی می‌کنند، استفاده از وقفه‌ی[۱۶۶] نخ است که باعث می‌گردد نخی که در خواب است، فوراً بیدار شود و وقفه را اداره نماید.

```
final class ControlledStop implements Runnable
{
	@Override public void run()
	{
		// Record current thread so others can interrupt it
		myThread = currentThread();
		while (!Thread.interrupted())
		{
			try
			{
				Thread.sleep(1000);
			}
			catch (InterruptedException e)
			{
				Thread.currentThread().interrupt();
			}
		}
	}
	public void shutdown(Thread th)
	{
		th.interrupt();
	}
}
```

توجه داشته‌باشید که نخ وقفه‌دهنده باید بداند برای کدام نخ، وقفه ایجاد نماید؛ منطق پیگیری این رابطه، از این راه‌حل حذف شده‌است.

۴,۲,۶ نمونه‌کد ناسازگار (*Thread.getState()*)

این نمونه‌کد، حاوی یک متد *doSomething()* است که یک نخ را آغاز می‌نماید. نخ، با بررسی یک پرچم و انتظار تا زمان اطلاع‌رسانی، از وقفه پشتیبانی می‌کند. متد *stop()* بررسی می‌نماید که آیا نخ، به انتظار،

[۱۶۶] Interruption

بلاک شده‌است یا خیر؛ اگر چنین است، به پرچم، مقدار *true* را داده و به نخ اطلاع می‌دهد تا نخ بتواند پایان یابد.

```
public class Waiter {
	private Thread thread;
	private boolean flag;
	private final Object lock = new Object();
	public void doSomething() {
		thread = new Thread(new Runnable() {
			@Override public void run() {
				synchronized (lock) {
					while (!flag) {
						try {
							lock.wait();
							// ...
						}
						catch (InterruptedException e) {
							// Forward to handler
						}
					}
				}
			}
		});
		thread.start();
	}
	public boolean stop() {
		if (thread != null) {
			if (thread.getState() == Thread.State.WAITING) {
				synchronized (lock) {
					flag = true;
					lock.notifyAll();
				}
				return true;
			}
		}
		return false;
	}
}
```

متاسفانه، متد *stop()* برای بررسی این‌که نخ بلاک شده‌است و پیش از ارسال اطلاع، خاتمه نیافته‌است، به‌صورت نادرست، از متد *Thread.getState()* استفاده می‌کند. استفاده از متد *Thread.getState()* برای کنترل همگام‌سازی، مانند بررسی این‌که آیا یک نخ در انتظار، بلاک شده‌است یا خیر، نامناسب است.

JVMها اجازه دارند با استفاده از انتظار چرخشی، بلاک کردن را پیاده‌سازی کنند. در نتیجه، یک نخ می‌تواند بدون وارد کردن وضعیت *WAITING* یا *TIMED_WAITING*، بلاک شود. به‌دلیل آن‌که ممکن است نخ، هرگز وارد وضعیت *WAITING* نشود، این احتمال وجود دارد که متد *stop()* در خاتمه بخشیدن به نخ، شکست بخورد.

اگر *doSomething()* و *stop()* از نخ‌های متفاوتی فراخوانی شوند، متد *stop()* می‌تواند در مقداردهی اولیه‌ی نخ شکست بخورد، حتی اگر *doSomething()* زودتر فراخوانی شود. مگر این‌که، یک رابطه‌ی از پیش رخداده‌ای بین دو فراخوانی وجود داشته‌باشد. اگر دو متد توسط یک نخ فراخوانی شوند، به‌صورت خودکار یک دارای رابطه‌ی از پیش رخداده هستند و در نتیجه، نمی‌توانند با این مشکل مواجه شوند.

۵,۲,۶ راه‌حل سازگار

این راه‌حل، بررسی برای تعیین این‌که آیا نخ در وضعیت *WAITING* قرار دارد یا خیر را حذف می‌کند. این بررسی، غیر ضروری است، زیرا فراخوانی *notifyAll()*، تنها بر نخ‌هایی که بر اثر فراخوانی *wait()* بلاک شده‌اند، اثرگذار است.

```
public class Waiter {
	// . . .
	private Thread thread;
	private volatile boolean flag;
	private final Object lock = new Object();
	public boolean stop() {
		if (thread != null) {
			synchronized (lock) {
				flag = true;
				lock.notifyAll();
			}
			return true;
		}
		return false;
	}
}
```

۶,۲,۶ کاربرد

تکیه بر متدهای *sleep()*، *yield()* و *getState()* متعلق به کلاس *Thread* جهت کنترل همگام‌سازی، می‌تواند منجر به رفتار غیرمنتظره شود.

۳,۶ فرض نکنید که همواره، عملگر باقی‌مانده، یک نتیجه‌ی غیرمنفی را برای عملوندهای صحیح بر می‌گرداند

عملگر باقی‌مانده، برای عملوندهایی که پس از ارتقای عددی باینری، عدد صحیح هستند، مقدار نتیجه‌ای تولید می‌کند که طبق آن، *(a/b)*b+(a%b)* برابر با *a* است. این تطابق، حتی در موارد خاصی که مقسوم بزرگترین عدد صحیح منفی ممکن در نوع خود بوده و مقسوم‌علیه مقدار ۱- را در اختیار دارد (که باقیمانده، ۰ می‌شود)، برقرار است. همچنین، از این قانون پیروی می‌کند که باقی‌مانده، تنها در صورتی می‌تواند منفی شود که مقسوم، منفی باشد؛ و تنها در صورتی می‌تواند مثبت باشد که مقسوم، مثبت باشد؛ علاوه بر این، نتیجه، همواره کوچکتر از مقسوم است. علامت باقی‌مانده همواره مشابه با مقسوم‌علیه (اولین عملوند در عبارت) است:

```
5 % 3 produces 2
5 % (-3) produces 2
(-5) % 3 produces -2
(-5) % (-3) produces -2
```

در نتیجه، نباید کد فرض کند که عملگر باقی‌مانده همواره یک مقدار مثبت را بر می‌گرداند.

۱,۳,۶ نمونه‌کد ناسازگار

این نمونه‌کد، از عدد صحیح *hashKey*، به‌عنوان اندیس آرایه‌ی *hash* استفاده می‌نماید:

```
private int SIZE = 16;
public int[] hash = new int[SIZE];
public int lookup(int hashKey)
{
      return hash[hashKey % SIZE];
}
```

یک کلید *hash*، یک نتیجه‌ی منفی برای عملگر باقی‌مانده تولید می‌کند و باعث می‌شود متد *lookup()*، استثنای *java.lang.ArrayIndexOutOfBoundsException* را ایجاد نماید.

۲,۳,۶ راه‌حل سازگار

این راه‌حل، متد *imod()* را فراخوانی می‌کند که همواره، یک باقیمانده‌ی مثبت را بر می‌گرداند:

```
// Method imod() gives nonnegative result
private int SIZE = 16;
public int[] hash = new int[SIZE];
private int imod(int i, int j)
{
	int temp = i % j;
	// Unary minus will succeed without overflow
	// because temp cannot be Integer.MIN_VALUE
	return (temp < 0) ? -temp : temp;
}
public int lookup(int hashKey)
{
return hash[imod(hashKey, SIZE)];
}
```

۳,۳,۶ کاربرد

فرض نادرست در مورد یک باقی‌مانده‌ی مثبت از یک عملگر باقی‌مانده، می‌تواند منجر به کد نادرست شود.

۴,۶ برابری شی انتزاعی را با برابری ارجاع اشتباه نگیرید

جاوا، عملگرهای برابری == و =! را برای آزمودن برابری ارجاع تعریف نمود، اما از متد *equals()* تعریف‌شده در *Object* و زیرکلاس‌های آن، برای آزمودن برابری شی انتزاعی استفاده می‌کند. برنامه‌نویسان بی‌تجربه اغلب هدف عملگر == را با *Object.equals()* اشتباه می‌گیرند. این سردرگمی، مکرراً در مفهوم پردازش اشیای *String* مشهود است.

به‌عنوان یک قانون کلی، از متد *Object.equals* برای بررسی برابری محتوای دو شی، و از عملگرهای == و =!، برای آزمودن این‌که دو مرجع، دقیقاً به یک *شی مشابه* ارجاع می‌کنند، استفاده کنید. به مورد آخر، برابری ارجاعی[۱۶۷] نیز اطلاق می‌شود. برای کلاس‌هایی که به بازنویسی پیاده‌سازی *equals()* پیش‌فرض نیاز دارند، باید مراقب بود که متد *hashCode()* نیز بازنویسی شود (MET09-J. کلاس‌هایی که یک متد *equals()* را تعریف می‌کنند، باید یک متد *hashCode()* را نیز تعریف نمایند).

انواع عددی باکس‌شده[۱۶۸] (مانند *Byte*، *Charachter*، *Short*، *Integer*، *Long*، *Float*، و *Double*) باید به جای عملگر ==، توسط *Object.equals()* نیز مقایسه شوند. در حالی‌که ممکن است برابری ارجاع، برای مقادیر صحیح بین ۱۲۸- تا ۱۲۷ کار کند، امکان دارد در صورتی‌که هر یک از عملوندهای موجود در مقایسه، خارج از محدوده باشند، شکست بخورد. عملگرهای رابطه‌ای عددی به‌جز برابری (مانند >، =>، < و =<) می‌توانند به‌طور امن برای مقایسه‌ی انواع باکس‌شده‌ی اصلی استفاده شوند (EXP03-J. از عملگرهای برابری هنگام مقایسه‌ی مقادیر باکس‌شده‌ی اصلی استفاده نکنید).

1.4.6 نمونه‌کد ناسازگار

این نمونه‌کد، دو شی *String* مجزا را که حاوی مقدار مشابهی هستند، اعلان می‌کند. عملگر برابری ارجاع ==، تنها هنگامی *true* ارزیابی می‌شود که مقادیر مورد مقایسه‌ی آن، به شی زیربنایی مشابهی اشاره نمایند. ارجاع‌ها در این مثال برابر نیستند، زیرا به اشیای مجزایی ارجاع می‌دهند.

---

[۱۶۷] Referential Equality

[۱۶۸] Numeric boxed types

```
public class StringComparison
{
	public static void main(String[] args)
	{
		String str1 = new String("one");
		String str2 = new String("one");
		System.out.println(str1 == str2); // Prints "false"
	}
}
```

۲,۴,۶ راه‌حل سازگار (*Object.equals()*)

این راه‌حل، هنگام مقایسه مقادیر رشته‌ای، از متد *Object.equals()* استفاده می‌کند:

```
public class StringComparison
{
	public static void main(String[] args)
	{
		String str1 = new String("one");
		String str2 = new String("one");
		System.out.println(str1.equals(str2)); // Prints "true"
	}
}
```

۳,۴,۶ راه‌حل سازگار (*String.intern()*)

هنگامی‌که برابری ارجاع برای مقایسه‌ی دو رشته‌ی حاصل از متد *String.intern()* استفاده می‌شود، همانند برابری شی انتزاعی عمل می‌کند. این راه‌حل، از *String.intern()* استفاده می‌نماید و می‌تواند هنگامی‌که تنها به یک کپی از رشته‌ی *one* در حافظه نیاز است، مقایسه‌ای سریع انجام دهد.

استفاده از *String.intern()* باید برای مواردی باشد که در آن، نشانه‌گذاری[۱۶۹] رشته‌ها به یک ارتقای کارایی مهم برسد یا کد را به‌شدت ساده کند. مثال‌هایی مانند برنامه‌هایی که درگیر پردازش زبان طبیعی هستند و ابزارهای مشابه با کامپایلر که ورودی برنامه را نشانه‌گذاری می‌کنند، در این زمره قرار می‌گیرند.

[۱۶۹] Tokenization

برای اکثر برنامه‌های دیگر، کارایی و قابلیت خوانایی، اغلب با استفاده از کدی که به رویکرد *Object.equals()* اعمال می‌شود و فاقد هرگونه وابستگی به برابری ارجاع است، بهبود می‌یابند.

```
public class StringComparison
{
	public static void main(String[] args)
	{
		String str1 = new String("one");
		String str2 = new String("one");
		str1 = str1.intern();
		str2 = str2.intern();
		System.out.println(str1 == str2); // Prints "true"
	}
}
```

JLS تضمین‌های کمی در مورد پیاده‌سازی *String.intern()* فراهم می‌سازد. به‌عنوان نمونه،

- هزینه‌ی *String.intern()* با افزایش تعداد رشته‌های داخلی، افزایش می‌یابد و کارایی نباید بدتر از *O(n log n)* باشد، اما JLS فاقد یک تضمین کارایی مشخص است.
- در پیاده‌سازی‌های اولیه‌ی JVM، رشته‌های داخلی، جاودانه می‌شوند: آنها از زباله‌روبی، معاف هستند. وقتی تعداد زیادی از رشته‌ها، داخلی باشند، مشکل‌ساز می‌گردد. پیاده‌سازی‌های جدیدتر، فضای ذخیره‌سازی اشغال‌شده توسط رشته‌های داخلی، که دیگر به آنها ارجاع داده نمی‌شود، می‌تواند زباله‌روبی شود. با این‌وجود، JLS، در مورد جزئیات و خصوصیات این رفتار توضیح نمی‌دهد.
- در پیاده‌سازی‌های JVM پیش از نسخه‌ی 1.7 جاوا، رشته‌های داخلی در ناحیه‌ی ذخیره‌سازی *permgen* تخصیص داده می‌شدند، که از بقیه‌ی هیپ، بسیار کوچکتر است. در نتیجه، داخلی‌سازی تعداد زیادی از رشته‌ها می‌تواند منجر به وضعیت اتمام حافظه[۱۷۰] شود. در بسیاری از پیاده‌سازی‌های نسخه‌ی 1.7 جاوا، رشته‌های داخلی در هیپ تخصیص داده‌می‌شدند، که محدودیت

[۱۷۰] Out-of-memory

مذکور را تسکین می‌دادند. همچنان، جزئیات تخصیص توسط JLS مشخص نشده‌اند. در نتیجه، ممکن است پیاده‌سازی‌ها متفاوت باشند.

ممکن است داخلی‌سازی رشته در برنامه‌هایی استفاده شوند که رشته‌ها در آنها مکررا رخ دهند. استفاده از آن منجر به ارتقای کارایی مقایسات و کمینه نمودن مصرف حافظه می‌شود.

هنگامی که به متعارف‌سازی[۱۷۱] اشیا نیاز است، استفاده از متعارف‌سازی که روی *ConcurrentHashMap* ساخته شده‌است، می‌تواند عاقلانه‌تر باشد.

۴.۴.۶ کاربرد

اشتباه گرفتن برابری ارجاع و برابری شی می‌تواند منجر به نتایج غیرمنتظره شود. استفاده از برابری ارجاع به جای برابری شی، تنها هنگامی مجاز است که تعریف کلاس‌ها، وجود حداکثر یک نمونه شی را برای هر مقدار ممکن شی، تضمین نماید. استفاده از متدهای تولیدگر ایستا به جای سازنده‌های عمومی، کنترل نمونه را تسهیل می‌سازد، که این امر، روشی کلیدی است. روش دیگر، استفاده از نوع *enum* است. از برابری ارجاع، برای تعیین این‌که آیا دو مرجع، به یک شی مشابه ارجاع می‌دهند یا خیر، استفاده کنید.

[۱۷۱] Canonicalization

۵,۶ تفاوت میان عملگرهای بیتی و منطقی را درک کنید

عملگرهای *AND* و *OR* (به‌ترتیب، && و ||) رفتار اتصال کوتاه[۱۷۲] از خود نشان می‌دهند. یعنی، عملوند دوم تنها زمانی ارزیابی می‌شود که عملگر شرطی نتواند به‌تنهایی توسط عملوند اول استنباط شود. در نتیجه، هنگامی‌که نتیجه‌ی عملگر شرطی *بتواند* به‌تنهایی از نتیجه‌ی عملوند اول استنباط شود، عملوند دوم، بدون ارزیابی باقی خواهد ماند. اثر جانبی آن، در صورت وجود، هرگز رخ نخواهد داد.

عملگرهای بیتی *AND* و *OR* (به‌ترتیب، & و |) فاقد رفتار اتصال کوتاه هستند. همانند با اکثر عملگرهای جاوا، هر دو عملوند را ارزیابی می‌کنند. به مثابه && و ||، نتایج *Boolean* تولید می‌نمایند، اما می‌توانند بسته به حضور یا عدم حضور اثرات جانبی در عملوند دوم، اثرات کلی متفاوتی داشته‌باشند.

در نتیجه، هر یک از عملگرهای & یا && می‌توانند هنگام انجام منطق *Boolean* مورد استفاده قرار گیرند. با این‌وجود، زمان‌هایی وجود دارد که رفتار اتصال کوتاه ترجیح داده‌می‌شود و در سایر زمان‌ها، رفتار اتصال کوتاه منجر به اشکالات نامحسوس می‌شود.

۱,۵,۶ نمونه‌کد ناسازگار (& نامناسب)

این نمونه‌کد، دارای دو متغیر با مقادیر ناشناخته است. کد باید داده‌ی خود را اعتبارسنجی کرده و سپس، اعتبار اندیس *array[i]* را بررسی کند.

```
int array[]; // May be null
int i; // May be an invalid index for array
if (array != null & i >= 0 & i < array.length & array[i] >= 0)
{
	// Use array
}
else
{
	// Handle error
}
```

[۱۷۲] Short-circuit

این کد، به‌دلیل همان خطاهایی که سعی می‌کند از آنها دوری کند، می‌تواند شکست بخورد. هنگامی‌که *array* برابر با *NULL* است یا *i* یک اندیس معتبر نیست، ارجاع به *array* و *array[i]*، منجر به استثنای *NullPointerException* یا *ArrayIndexOutOfBoundsException* خواهد شد. استثنا به این دلیل رخ می‌دهد که عملگر & در جلوگیری از ارزیابی عملوند سمت راست خود، حتی زمانی‌که ارزیابی عملوند سمت چپ آن ثابت می‌کند که عملوند راست بی‌اهمیت است، شکست می‌خورد.

۲٫۵٫۶ راه‌حل سازگار (استفاده از &&)

این راهحل، مشکل را با استفاده از && کاهش می‌دهد، که در صورت شکست هر یک از شرط‌ها، منجر به خاتمه‌ی فوری ارزیابی عبارت شرطی می‌شود و در نتیجه، جلوی یک استثنا زمان اجرا را می‌گیرد:

```
int array[]; // May be null
int i; // May be an invalid index for array
if (array != null && i >= 0 && i < array.length && array[i] >= 0)
{
	// Handle array
}
else
{
	// Handle error
}
```

۳٫۵٫۶ راه‌حل سازگار (عبارت‌های *if* تو در تو)

این راه‌حل، از چندین عبارت *if* جهت دست‌یابی به اثر مناسب، استفاده می‌کند. اگرچه این راه‌حل، طولانی‌تر است و نگهداری از آن می‌تواند دشوارتر باشد. با این‌وجود، هنگامی‌که کد اداره‌ی خطا برای هر حالت شکست احتمالی، متفاوت است، ترجیح داده می‌شود.

```
int array[]; // May be null
int i; // May be a valid index for array
if (array != null) {
	if (i >= 0 && i < array.length)
	{
		if (array[i] >= 0)
		{
			// Use array
		}
		else
		{
			// Handle error
		}
	}
	else
	{
		// Handle error
	}
}
else
{
	// Handle error
}
```

۴,۵,۶ نمونه‌کد ناسازگار (&& نامناسب)

این نمونه‌کد، کدی را نشان می‌دهد که دو آرایه را برای بازه‌ای از اعضا، که با هم تطبیق دارند، مقایسه می‌کند. در این‌جا، *i1* و *i2* به‌ترتیب، اندیس‌های معتبر آرایه‌های *array1* و *array2* هستند. متغیرهای *end1* و *end2* نیز، اندیس‌های انتهای بازه‌های تطبیقی در دو آرایه هستند.

```
if (end1 >= 0 & i2 >= 0)
{
	int begin1 = i1;
	int begin2 = i2;
	while(++i1<array1.length && ++i2<array2.length &&array1[i1]==array2[i2])
	{
		// Arrays match so far
	}
	int end1 = i1;
	int end2 = i2;
	assert end1 - begin1 == end2 - begin2;
}
```

مشکل این کد آن است که وقتی شرط اول در حلقه‌ی *while* شکست می‌خورد، شرط دوم اجرا نمی‌شود؛ یعنی، به‌محض این‌که *i2* به *array1.length* رسید، حلقه، پس از افزایش *i1*، خاتمه می‌یابد. در نتیجه، بازه‌ی ظاهری *array1* بزرگتر از بازه‌ی ظاهری *array2* است، که منجر به شکست اعلان نهایی می‌شود.

۵,۵,۶ راه‌حل سازگار (استفاده از &)

این راه‌حل، مشکل را با استفاده از & و به صورت محتاطانه کاهش می‌دهد، که تضمین می‌کند هر دوی *i1* و *i2*، صرف‌نظر از پیامد شرط اول، افزایش یافته‌اند.

```
public void exampleFuntion()
{
	while(++i1<array1.length & // Not && ++i2<array2.length &&
		array1[i1] == array2[i2])
		{
			// Do something
		}
}
```

۶,۵,۶ کاربرد

شکست در درک رفتار عملگرهای بیتی و شرطی، می‌تواند منجر به رفتار ناخواسته‌ی برنامه شود.

۶,۶ درک کنید که کاراکترهای *escape*، چگونه هنگام بارگذاری رشته‌ها تفسیر می‌شوند

بسیاری از کلاس‌ها، اجازه‌ی گنجاندن دنباله‌های *escape* (ملقب به دنباله‌های *ESC*، رشته‌های خاصی از کدهای کاراکتری که موجب می‌شوند یک صفحه نمایش یا چاپ‌گر، به جای نمایش کاراکترها، کار خاصی را انجام دهد) را در لفظ‌های کاراکتر و رشته، می‌دهند. مثال‌ها، شامل *java.util.regex.Pattern* و کلاس‌هایی که از عملیات مبتنی بر XML و SQL با ارسال پارامترهای رشته‌ای به متدها پشتیبانی می‌کنند، هستند. کاراکتر و رشته‌ی نمایش برخی از کاراکترهای غیرگرافیکی، ‘، ”، \ در لفظ‌های کاراکتری و لفظ‌های رشته‌ای، در دنباله‌های *escape*، مجاز هستند.

استفاده‌ی صحیح از دنباله‌های *escape* در لفظ‌های رشته‌ای، به درک چگونگی تفسیر دنباله‌های *ESC* توسط کامپایلر جاوا و این‌که چگونه توسط یک پردازنده‌ی متعاقب، مانند موتور SQL، تفسیر می‌گردند، نیاز دارد. ممکن است عبارات SQL، به دنباله‌های *escape* (مانند دنباله‌هایی حاوی *t*\، *n*\، و *r*\) در موارد خاصی، مانند ذخیره‌ی متن خام در پایگاه داده، نیاز دارند. هر دنباله‌ی *ESC* باید هنگام نمایش عبارت‌های SQL در لفظ‌های رشته‌ای جاوا، با یک \ اضافی برای تفسیر درست، آغاز شود.

به‌عنوان مثالی دیگر، کلاس *Pattern* را در انجام وظایف مرتبط با عبارت منظم درنظر بگیرید. یک لفظ رشته‌ای مورد استفاده برای تطبیق الگو، به یک نمونه از نوع *Pattern* کامپایل می‌شود. وقتی الگویی که باید تطبیق داده شود، حاوی دنباله‌ای از کاراکترهای یکسان با دنباله‌های *ESC* جاوا باشد، مانند \ و *n*، کامپایلر جاوا با آن بخش از رشته، به‌عنوان یک دنباله‌ی *ESC* جاوا برخورد می‌کند و دنباله را به یک کاراکتر خط جدید واقعی، تبدیل می‌نماید. برنامه‌نویس باید برای افزودن یک دنباله‌ی *ESC* خط جدید به جای یک کاراکتر خط جدید لفظی، دنباله‌ی ”*n*\“ را با یک \ اضافی آغاز کند تا مانع از جایگزینی آن با یک کاراکتر خط جدید توسط کامپایلر جاوا شود. رشته‌ی ساخته شده از دنباله‌ی حاصل (”*n*\\“)، متعاقباً حاوی دنباله‌ی دو کاراکتری درست *n*\ خواهد بود و دنباله‌ی *ESC* را به‌درستی برای خط جدید در الگو، نشان می‌دهد.

به‌طور کلی، برای یک کاراکتر *ESC* مشخص به شکل *X*\، نمایش معادل جاوا به شکل زیر خواهد بود.

```
\\X
```

۱,۶,۶ نمونه‌کد ناسازگار (لفظ رشته)

این نمونه‌کد، یک متد *splitWords()* را تعریف می‌کند که تطابق‌های بین لفظ رشته‌ای (*WORDS*) و دنباله‌ی ورودی را پیدا می‌نماید. انتظار می‌رود که *WORDS*، دنباله‌ی *ESC* را برای تطابق یک مرز کلمه، نگه دارد. با این‌حال، کامپایلر جاوا با لفظ "\b"، به‌عنوان یک دنباله‌ی *ESC* جاوا برخورد می‌کند و رشته‌ی *WORDS*، در سکوت، به یک عبارت منظم کامپایل می‌شود که یک کاراکتر \ منفرد را بررسی می‌نماید.

```
public class Splitter
{
	// Interpreted as backspace
	// Fails to split on word boundaries
	private final String WORDS = "\b";
	public String[] splitWords(String input)
	{
		Pattern pattern = Pattern.compile(WORDS);
		String[] input_array = pattern.split(input);
		return input_array;
	}
}
```

۲,۶,۶ راه‌حل سازگار (لفظ رشته)

این راه‌حل، مقدار لفظ رشته‌ای *WORDS* را، که به‌درستی *ESC* شده‌است، نشان می‌دهد که منجر به یک عبارت منظم طراحی‌شده برای جداسازی مرزهای کلمه می‌شود:

```
public class Splitter
{
	// Interpreted as two chars, '\' and 'b'
	// Correctly splits on word boundaries
	private final String WORDS = "\\b";
	public String[] split(String input)
	{
		Pattern pattern = Pattern.compile(WORDS);
		String[] input_array = pattern.split(input);
	return input_array;
	}
}
```

۳,۶,۶ نمونه‌کد ناسازگار (ویژگی رشته)

این نمونه‌کد، از همان متد *splitWords()* استفاده می‌کند. این‌بار رشته‌ی *WORDS*، از یک فایل ویژگی‌های خارجی بارگذاری شده‌است.

```
public class Splitter
{
	private final String WORDS;
	public Splitter() throws IOException
	{
		Properties properties = new Properties();
		properties.load(new FileInputStream("splitter.properties"));
		WORDS = properties.getProperty("WORDS");
	}
	public String[] split(String input)
	{
		Pattern pattern = Pattern.compile(WORDS);
		String[] input_array = pattern.split(input);
		return input_array;
	}
}
```

در این فایل ویژگی‌ها، ویژگی *WORD* یک‌بار دیگر به نادرستی، به‌عنوان *\b* مشخص شده‌است.

```
WORDS=\b
```

این دستور به‌عنوان یک کاراکتر منفرد *b*، توسط متد *Properties.load()* خوانده می‌شود که باعث می‌گردد متد *split()*، رشته‌ها را روی حرف *b* جدا کند. اگرچه، مانند نمونه‌کد ناسازگار قبلی، رشته، متفاوت از حالتی که یک لفظ رشته‌ای منفرد بود، تفسیر می‌گردد، اما این تفسیر، نادرست است.

۴,۶,۶ راه‌حل سازگار (ویژگی رشته)

این راه‌حل، مقداری را که به‌درستی برای ویژگی *WORDS*، *ESC* شده‌است، نشان می‌دهد:

```
WORDS=\\b
```

۵,۶,۶ کاربرد

استفاده‌ی نادرست از کاراکترهای *ESC* در ورودی‌های رشته‌ای، می‌تواند منجر به تفسیر نادرست و احتمالاً، خرابی داده‌ها شود.

۷,۶ از متدهای سربار برای تمایز بین انواع زمان‌اجرا استفاده نکنید

جاوا، از متدهای حاوی سربار پشتیبانی می‌کند و می‌تواند بین متدهای با امضاهای متفاوت، تمایز قائل شود. در نتیجه، در صورتی‌که متدهای درون کلاس دارای لیست پارامترهای مختلفی باشند، می‌توانند تحت برخی شرایط[۱۷۳]، از نام یکسانی برخوردار گردند. در سربارسازی متد، متدی که باید در زمان اجرا فراخوانی شود، در زمان کامپایل تعیین می‌گردد. در نتیجه، حتی هنگامی‌که نوع زمان‌اجرا برای هر فراخوانی متفاوت باشد، متد حاوی سربار و مرتبط با نوع ایستای شی، فراخوانی می‌گردد.

۱,۷,۶ نمونه‌کد ناسازگار

این نمونه‌کد، تلاش می‌کند تا از متد حاوی سربار *display()*، برای انجام عملیات متفاوت (بسته به این‌که آیا متد یک *ArrayList<Integer>* یا یک *LinkedList<String>* را فرستاده است) استفاده نماید:

```
public class Overloader {
	private static String display(ArrayList<Integer> arrayList) {
		return "ArrayList";
	}
	private static String display(LinkedList<String> linkedList) {
		return "LinkedList";
	}
	private static String display(List<?> list) {
		return "List is not recognized";
	}
	public static void main(String[] args) {
		// Single ArrayList
		System.out.println(display(new ArrayList<Integer>()));
		// Array of lists
		List<?>[] invokeAll = new List<?>[] {
			new ArrayList<Integer>(),
			new LinkedList<String>(),
			new Vector<Integer>()
		};
		for (List<?> list : invokeAll) {
			System.out.println(display(list));
		}
	}
}
```

[۱۷۳] Qualifications

آرایه‌ی شی در زمان کامپایل، از نوع *List* است. خروجی مورد انتظار *ArrayList*، یکی از سه نوع *ArrayList*، *LinkedList* و *List is not recognized* است، زیرا *java.util.Vector* نه یک *ArrayList* است و نه یک *LinkedList*. خروجی واقعی، *ArrayList* است که توسط *List is not recognized*، که سه بار تکرار شده‌است، دنبال می‌گردد. علت این رفتار غیرمنتظره، این است که فراخوانی متد حاوی سربار، *تنها* توسط نوع زمان کامپایل پارامتر آنها، تحت تاثیر قرار می‌گیرند: *ArrayList*، برای اولین فراخوانی و *List*، برای سایرین.

۲,۷,۶ راه‌حل سازگار

این راه‌حل، از یک متد *display* و *instanceof* منفرد، برای تمایز بین انواع مختلف استفاده می‌کند. همان‌گونه که انتظار می‌رفت، خروجی *ArrayList*، یکی از سه نوع *ArrayList*، *LinkedList*، *List is not recognized* است:

```
public class Overloader {
	private static String display(List<?> list) {
		return (list instanceof ArrayList ? "Arraylist" :
			(list instanceof LinkedList ? "LinkedList" :
			"List is not recognized")
			);
	}
	public static void main(String[] args) {
		// Single ArrayList
		System.out.println(display(new ArrayList<Integer>()));
		List<?>[] invokeAll = new List<?>[] {
			new ArrayList<Integer>(),
			new LinkedList<String>(),
			new Vector<Integer>()
		};
		for (List<?> list : invokeAll) {
			System.out.println(display(list));
		}
	}
}
```

۳,۷,۶ کاربرد

استفاده‌های مبهم از سربار، می‌تواند منجر به نتایج غیرمنتظره شود.

۸,۶ هرگز تغییرناپذیری یک مرجع را با تغییرناپذیری شی مورد ارجاع، اشتباه نگیرید

تغییرناپذیری، به پشتیبانی از استدلال امنیتی کمک می‌کند. اشتراک اشیای تغییرناپذیر فارغ از این‌که دریافت‌کننده، تغییر را به خطر بیندازد، امن است.

برنامه‌نویسان، اغلب فرض می‌کنند که اعلان یک فیلد یا متغیر به‌عنوان *final*، شی مورد ارجاع را تغییرناپذیر می‌سازد. اعلان متغیرهایی با نوع اصلی به‌عنوان *final*، به‌وسیله‌ی پردازش طبیعی جاوا، از تغییرات مقادیر آنها پس از مقداردهی اولیه، جلوگیری می‌کند. با این‌حال، هنگامی‌که متغیر از نوع مرجع باشد، حضور عبارت *final* در اعلان، تنها خود ارجاع را تغییرناپذیر می‌کند. عبارت *final*، اثری بر شی مورد ارجاع ندارد. در نتیجه، ممکن است فیلدهای شی مورد ارجاع، تغییرپذیر شوند.

اگر یک متغیر *final* ، یک ارجاع به یک شی را نگه دارد، آن‌گاه، ممکن است وضعیت شی توسط عملیات روی شی، تغییر یابد، اما متغیر همواره به همان شی، ارجاع خواهد کرد. این امر، حتی به آرایه‌ها نیز اعمال می‌گردد، زیرا آرایه‌ها، اشیا هستند؛ اگر یک متغیر *final*، ارجاعی به یک آرایه داشته‌باشد، آن‌گاه، ممکن است مولفه‌های آرایه هنوز توسط عملیات روی آرایه تغییر کنند، اما متغیر همواره به همان آرایه ارجاع خواهد داشت.

به‌طور مشابه، یک پارامتر متد *final*، یک کپی غیرقابل تغییری از *ارجاع شی* نگه می‌دارد. مجدداً، این کار اثری بر تغییرپذیری داده‌ی مورد ارجاع ندارد.

۱,۸,۶ نمونه‌کد ناسازگار (کلاس تغییرپذیر، ارجاع *final*)

در این نمونه‌کد، برنامه‌نویس، ارجاع به نمونه‌ی *point* را *final* اعلان می‌کند که تحت این فرضیه، نادرست است و انجام چنین کاری از اصلاحات *مقادیر* متغیرهای نمونه *x* و *y*، ممانعت به‌عمل می‌آورد. مقادیر متغیرهای نمونه می‌توانند پس از مقداردهی اولیه به آنها، تغییر یابند، زیرا عبارت *final*، تنها به ارجاع به نمونه‌ی *point* اعمال می‌شود و نه به شی مورد ارجاع.

```
class Point {
	private int x;
	private int y;
	Point(int x, int y) {
		this.x = x;
		this.y = y;
	}
	void set_xy(int x, int y) {
		this.x = x;
		this.y = y;
	}
	void print_xy() {
		System.out.println("the value x is: " + this.x);
		System.out.println("the value y is: " + this.y);
	}
}
public class PointCaller {
	public static void main(String[] args) {
		final Point point = new Point(1, 2);
		point.print_xy();
		// Change the value of x, y
		point.set_xy(5, 6);
		point.print_xy();
	}
}
```

۲,۸,۶ راه‌حل سازگار (فیلدهای *final*)

هنگامی‌که مقادیر متغیرهای نمونه‌ی x و y، پس از مقداردهی اولیه باید بدون تغییر باقی بمانند، باید به صورت *final* اعلان شوند. با این‌وجود، این امر، متد *set_xy()* را نامعتبر می‌سازد، زیرا دیگر نمی‌تواند مقدار x و y را تغییر دهد:

```
class Point {
	private final int x;
	private final int y;
	Point(int x, int y) {
		this.x = x;
		this.y = y;
	}
	void print_xy() {
		System.out.println("the value x is: " + this.x);
		System.out.println("the value y is: " + this.y);
	}
// set_xy(int x, int y) no longer possible
```

با این اصلاح، مقادیر متغیرهای نمونه، تغییرناپذیر می‌شوند و در نهایت، با مدل استفاده‌ی موردنظر برنامه‌نویس مطابقت پیدا می‌کنند.

۳.۸.۶ راه‌حل سازگار (عمل کپی را فراهم کنید)

اگر لازم است کلاس، تغییرپذیر باقی بماند، باید راه‌حل سازگار دیگری برای عمل کپی فراهم شود. این راه‌حل، یک متد *clone()* را در کلاس *Point* فراهم می‌کند، که از حذف متد آراینده[۱۷۴]، جلوگیری به‌عمل می‌آورد.

متد *clone()*، یک کپی از شی اصلی، که منعکس‌کننده‌ی وضعیت شی اصلی در زمان کلون‌سازی است، برمی‌گرداند. این شی جدید می‌تواند بدون در معرض قرار دادن شی اصلی، استفاده شود. از آنجاکه متد فراخوانی‌کننده، تنها ارجاع به نمونه‌ی کلون‌شده‌ی جدید را نگه می‌دارد، متغیرهای نمونه نمی‌توانند بدون همکاری فراخوانی‌کننده، تغییر یابند. این استفاده از متد *clone()*، به کلاس اجازه می‌دهد تا به‌صورت امن، قابل تغییر باقی بماند. (OBJ04-J. عمل کپی را برای کلاس‌های تغییرپذیر فراهم کنید تا اجازه‌ی ارسال نمونه‌ها را به‌صورت امن، به کد غیرقابل اعتماد بدهد).

کلاس *Point*، به‌منظور جلوگیری از بازنویسی متد *clone()* توسط زیرکلاس‌ها، به‌صورت *final* اعلان شده‌است. این امر، کلاس را قادر می‌سازد تا به‌صورت مناسب و بدون هیچ تغییر غیرعمدی در شی اصلی، استفاده گردد.

[۱۷۴] Setter

```
final public class Point implements Cloneable {
	private int x;
	private int y;
	Point(int x, int y) {
		this.x = x;
		this.y = y;
	}
	void set_xy(int x, int y) {
		this.x = x;
		this.y = y;
	}
	void print_xy() {
		System.out.println("the value x is: "+ this.x);
		System.out.println("the value y is: "+ this.y);
	}
	public Point clone() throws CloneNotSupportedException {
		Point cloned = (Point) super.clone();
		// No need to clone x and y as they are primitives
		return cloned;
	}
}
public class PointCaller {
	public static void main(String[] args)
	throws CloneNotSupportedException {
		Point point = new Point(1, 2); // Is not changed in main()
		point.print_xy();
		// Get the copy of original object
		Point pointCopy = point.clone();
		// pointCopy now holds a unique reference to the newly cloned Point
		// instance Change the value of x,y of the copy
		pointCopy.set_xy(5, 6);
		// Original value remains unchanged
		point.print_xy();
	}
}
```

۴,۸,۶ نمونه‌کد ناسازگار (آرایه‌ها)

این نمونه‌کد، از *items* یک آرایه‌ی *public static final* استفاده می‌کند:

```
public static final String[] items = {/* . . . */};
```

مشتریان می‌توانند به‌طور جزئی، محتوای آرایه را اصلاح نمایند. اگرچه، اعلان آرایه‌ی مرجع به‌عنوان *final*، جلوی اصلاح خود ارجاع را می‌گیرد.

۵,۸,۶ راه‌حل سازگار (دریافت‌کننده‌ی اندیس)

این راه‌حل، آرایه را *private* می‌کند و متدهای *public* را فراهم می‌سازد تا آیتم‌های منفرد و سایز آرایه را به‌دست آورند:

```
private static final String[] items = {/* . . . */};
public static final String getItem(int index)
{
	return items[index];
}
public static final int getItemCount()
{
	return items.length;
}
```

فراهم‌سازی دسترسی مستقیم به خود اشیای آرایه، امن است، زیرا *String*، تغییرناپذیر است.

۶,۸,۶ راه‌حل سازگار (آرایه را کلون کنید)

این راه‌حل، یک آرایه‌ی *private* و یک متد *public* را تعریف می‌کند که یک کپی از آرایه را برمی‌گرداند:

```
private static final String[] items = {/* . . . */};
public static final String[] getItems()
{
	return items.clone();
}
```

چون یک کپی از آرایه بازگردانده می‌شود، مقادیر اصلی آرایه نمی‌توانند توسط یک مشتری تغییر یابند. توجه داشته‌باشید که یک کپی عمیق دستی[۱۷۵] می‌تواند هنگام برخورد با اشیای آرایه‌ای، مورد نیاز باشد. این امر، عموماً زمانی اتفاق می‌افتد که اشیا، یک متد *clone()* را صدور نمی‌کنند (OBJ06-J. ورودی‌های قابل تغییر و مولفه‌های داخلی قابل تغییر را به‌صورت تدافعی کپی کنید).

[۱۷۵] Manual deep copy

همانند قبل، این متد، دسترسی مستقیم به خود اشیای آرایه‌ای را پیدا می‌کند؛ اما این‌کار، امن است، زیرا *String*، تغییرناپذیر است. اگر آرایه، حاوی اشیای تغییرپذیر باشد، متد *getItems()* می‌تواند به جای آن، آرایه‌ای از اشیای کلون‌شده را برگرداند.

۷.۸.۶ راه‌حل سازگار (پنهان‌سازی‌های غیرقابل اصلاح)

این راه‌حل، یک آرایه‌ی *private* را اعلان می‌کند که از آن، یک لیست غیرقابل تغییر *public* ساخته می‌شود:

```
private static final String[] items = {/* . . . */};
public static final List<String> itemsList =
      Collections.unmodifiableList(Arrays.asList(items));
```

هیچ‌یک از مقادیر اصلی آرایه یا لیست *public*، نمی‌توانند توسط یک مشتری اصلاح شوند (برای کلاس‌های تغییرپذیر حساس، پنهان‌سازی‌های غیرقابل اصلاح فراهم کنید). این راه‌حل می‌تواند هنگامی‌که آرایه، حاوی اشیای تغییرپذیر است نیز، استفاده شود.

۸.۸.۶ کاربرد

این فرض نادرست که مراجع *final*، منجر به تغییرپذیر ماندن محتوای شی مورد ارجاع می‌شوند، می‌تواند منجر به این گردد که یک مهاجم، آن شی‌ای را تغییر دهد که باور می‌رفت غیرقابل تغییر است.

۹,۶ از متدهای سریال‌سازی *writeUnshared()* و *readUnshared()*، با احتیاط استفاده کنید

هنگامی‌که اشیا توسط متد *writeObject()*، سریال‌سازی می‌شوند، هر شی، تنها یک‌بار در جریان[۱۷۶] خروجی نوشته می‌شود. فراخوانی بار دوم متد *writeObject()* روی همان شی، یک مرجع پشتی[۱۷۷] برای نمونه‌ی سریال‌شده‌ی قبلی موجود در جریان، قرار می‌دهد. به‌همین صورت، متد *readObject()*، حداکثر یک نمونه را برای هر شی موجود در جریان ورودی، که قبلا توسط *writeObject()* نوشته شده‌بود، تولید می‌کند.

متد *writeUnshared()*، یک شی "به اشتراک گذاشته‌نشده" را برای *ObjectOutputStream* تولید می‌کند. این متد، با *writeObject* یکسان است؛ مگر در موردی‌که همواره، شی داده‌شده را، به‌عنوان یک شی جدید و منحصر به‌فرد، در جریان می‌نویسد (بر خلاف مرجع پشتی، که به یک نمونه‌ی سریال‌شده‌ی قبلی اشاره می‌کند).

به‌طور مشابه، متد *readUnshared()*، یک شی "به اشتراک گذاشته‌نشده" را از *ObjectInputStream* می‌خواند. این متد، با *readObject* یکسان است؛ به جز این‌که، از فراخوانی‌های بعدی به *readObject* و *read-Unshared* جهت برگرداندن ارجاعات اضافی به نمونه‌ی به‌دست‌آمده از این فراخوانی، که از سریال‌سازی خارج شده‌است، ممانعت به‌عمل می‌آورد.

در نتیجه، متدهای *writeUnshared()* و *readUnshared()* برای سریال‌سازی رفت و برگشتی ساختمان‌های داده‌ای که حاوی چرخه‌های مرجع هستند، نامناسب هستند. نمونه‌کد زیر را درنظر بگیرید:

---

[۱۷۶] Stream

[۱۷۷] Back-reference

```
public class Person {
	private String name;
	Person() {
		// Do nothing – needed for serialization
	}
	Person(String theName) {
		name = theName;
	}
	// Other details not relevant to this example
}
public class Student extends Person implements Serializable {
	private Professor tutor;
	Student() {
		// Do nothing – needed for serialization
	}
	Student(String theName, Professor theTutor) {
		super(theName);
		tutor = theTutor;
	}
	public Professor getTutor() {
		return tutor;
	}
}
public class Professor extends Person implements Serializable {
	private List<Student> tutees = new ArrayList<Student>();
	Professor() {
		// Do nothing – needed for serialization
	}
	Professor(String theName) {
		super(theName);
	}
	public List<Student> getTutees () {
		return tutees;
	}
	/**
	*checkTutees checks that all the tutees have this Professor as their tutor
	*/
	public boolean checkTutees () {
		boolean result = true;
		for (Student stu: tutees) {
			if (stu.getTutor() != this) {
				result = false;
				break;
			}
		}
		return result;
	}
}
```

```
...
Professor jane = new Professor("Jane");
Student able = new Student("Able", jane);
Student baker = new Student("Baker", jane);
Student charlie = new Student("Charlie", jane);
jane.getTutees().add(able);
jane.getTutees().add(baker);
jane.getTutees().add(charlie);
System.out.println("checkTutees returns: " + jane.checkTutees());
// Prints "checkTutees returns: true"
```

Professor و *Student*، انواعی هستند که نوع پایه‌ای *Person* را توسعه می‌دهند. یک دانش‌آموز (که یک شی از نوع *Student* است) آموزگاری از نوع *Professor* دارد. یک پروفسور (که یک شی از نوع *Professor* است)، دارای لیستی (یک *ArrayList*) از شاگردانی (از نوع *Student*) است. متد *checkTutees()* بررسی می‌کند که آیا تمام شاگردان این پروفسور، او را به‌عنوان آموزگار خود در اختیار دارند یا خیر. اگر چنین بود، *true* و در غیر این‌صورت، *false* را برمی‌گرداند.

فرض کنید که *Professor Jane* سه دانش‌آموز به نام‌های *Able*، *Baker* و *Charlie* دارد که تمام آنها او را به‌عنوان آموزگار خود، دارند. مسائل هنگامی به‌وجود می‌آیند که متدهای *writeUnshared()* و *readUnshared()*، همانند نمونه‌کد ناسازگار زیر، با این کلاس‌ها استفاده شوند.

1.9.6 نمونه‌کد ناسازگار

این نمونه‌کد، تلاش می‌کند تا داده را با استفاده از *writeUnshared()*، سریال سازد. با این‌وجود، هنگامی‌که داده با استفاده از *readUnshared()*، از سریال‌سازی خارج شود، متد *checkTutees()*، دیگر مقدار *true* را برنمی‌گرداند، زیرا اشیای آموزگار سه دانش‌آموز، متفاوت از شی *Professor* اصلی هستند.

```
String filename = "serial";
	try (ObjectOutputStream oos = new ObjectOutputStream(new
	FileOutputStream(filename))) {
		// Serializing using writeUnshared
		oos.writeUnshared(jane);
	}
	catch (Throwable e) {
		// Handle error
	}
	// Deserializing using readUnshared
	try (ObjectInputStream ois = new ObjectInputStream(new
	FileInputStream(filename))){
		Professor jane2 = (Professor)ois.readUnshared();
		System.out.println("checkTutees returns: "+jane2.checkTutees());
	}
	catch (Throwable e) {
		// Handle error
	}
```

۲,۹,۶ راه‌حل سازگار

این راه‌حل، برای حصول اطمینان از این‌که شی آموزگار (که توسط سه دانش‌آموز مورد ارجاع است)، با شی *Professor* نگاشت یک‌به‌یک اصلی دارد، از متدهای *writeObject()* و *readObject()* استفاده می‌کند. متد *checkTutees()*، به‌درستی، *true* را برمی‌گرداند.

```
String filename = "serial";
try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(filename))) {
	// Serializing using writeUnshared
	oos.writeObject(jane);
}
catch (Throwable e) {
	// Handle error
}
// Deserializing using readUnshared
try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(filename))) {
	Professor jane2 = (Professor)ois.readObject();
	System.out.println("checkTutees returns: " + jane2.checkTutees());
}
catch (Throwable e) {
	// Handle error
}
```

۳.۹.۶ کاربرد

ممکن است هنگامی‌که ساختارهای داده‌ای حاوی چرخه‌های مرجع برای سریال‌سازی رفت و برگشتی استفاده می‌شوند، به کارگیری متدهای *writeUnshared()* و *readUnshared()*، نتایج غیرمنتظره‌ای را تولید کند.

۱۰,۶ سعی نکنید با تخصیص مقدار *null* به متغیرهای مرجع محلی، به زباله‌روب (GC) کمک کنید

تخصیص مقدار *null* به متغیرهای مرجع محلی، جهت "کمک به GC"، غیرضروری است. این‌کار، منجر به افزایش درهم‌ریختگی کد می‌شود و می‌تواند نگهداری را دشوار سازد. کامپایلرهای JIT جاوا[۱۷۸] و بیشتر پیاده‌سازی‌ها می‌توانند یک تجزیه و تحلیل بقای معادل را انجام دهند.

یک رویکرد بد مرتبط، استفاده از یک پایان‌بخش برای *null* کردن مراجع است (MET12-J. از پایان‌بخش‌ها استفاده نکنید).

1.10.6 نمونه‌کد ناسازگار

در این نمونه‌کد، *buffer* یک متغیر محلی است که مرجعی به یک آرایه‌ی موقت دارد. برنامه‌نویس تلاش می‌کند تا با انتساب *null* به آرایه‌ی *buffer* (هنگامی‌که دیگر مورد نیاز نیست)، به GC کمک کند.

```
{
        // Local scope
        int[] buffer = new int[100];
        doSomething(buffer);
        buffer = null;
}
```

۲,۱۰,۶ راه‌حل سازگار

منطق برنامه، به کنترل سخت روی طول عمر یک شی مورد ارجاع از یک متغیر محلی، نیاز دارد. در موارد نامتعارف، که چنین کنترلی ضروری است، از یک بلاک لغوی[۱۷۹] برای محدود کردن حوزه‌ی متغیر استفاده کنید، زیرا هنگامی‌که از حوزه خارج شود، GC بلافاصله می‌تواند شی را جمع‌آوری کند. این راه‌حل، از یک بلاک لغوی برای کنترل طول عمر شی *buffer* استفاده می‌کند:

```
{
        // Limit the scope of buffer
        int[] buffer = new int[100];
        doSomething(buffer);
}
```

[۱۷۸] Java just-in-time compilers

[۱۷۹] Lexical

۳,۱۰,۶ کاربرد

وقتی در یک تلاش اشتباه برای کمک به GC جهت پس گرفتن حافظه‌ی مرتبط، به متغیرهای مرجع محلی نیازی نیست، تنظیم مقدار آنها به *null*، امری غیر ضروری به‌شمار می‌رود.

مراجع

[1]. F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, D. Svoboda, “Java™ Coding Guidelines: 75 Recommendations for Reliable and Secure Programs,” Pearson Education, 2014.

[2]. J. Jaworski, P. Perrone, V. S. R. K. Chaganti, “Java Security Handbook,” Sams Publishing, 2000.