www.javaspektrum.de

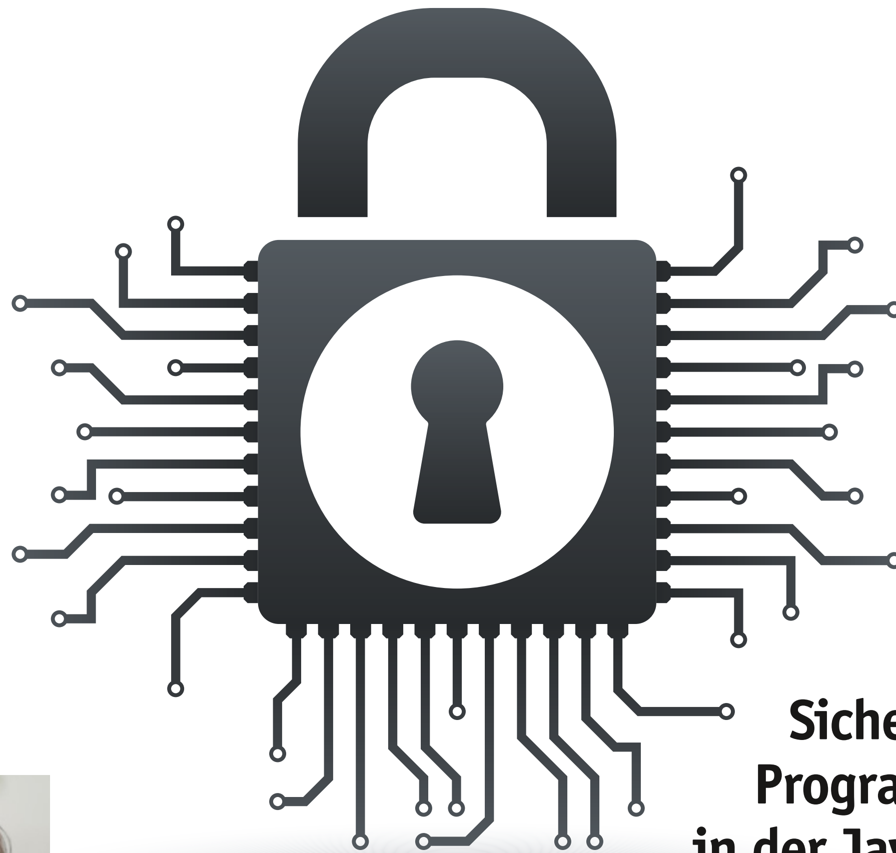
Ausgabe 5/2021
D € 9,90 · A € 10,90 · CHF 18,20
4 194156 109901 05

JavaSPEKTRUM

Magazin für professionelle Entwicklung und digitale Transformation

Einfach sicher – Sicherheitskonzepte in der Praxis

Einfach sicher – Sicherheitskonzepte in der Praxis



Sicheres Programmieren in der Java SE

Effektive, kostenfreie Security mit „FOSS"

Interview

Dr. Anke Sax von KGAL über die Herausforderungen einer Digitalisierungsstrategie und was die IT zur Pandemiebekämpfung beitragen kann

Fachthemen

Qualitätsstandards in die Agile Arbeitsweise übersetzen

Jakarta EE öffnet neue Türen für Cloud Native Java for Enterprise

zum Inhalt

G30325

Sicheres Java sicher belassen

Eine Orientierungshilfe für sicheres Programmieren in der Java SE

Christian Heitzmann

Entwickler und Anwender von Java-Software sind in der Regel vor vielen Sicherheitsproblemen gefeit, die in anderen Programmiersprachen noch problematisch sind. Doch unbedachte Code-Stellen können auch in Java schnell sicherheitsrelevante Löcher aufreißen. Dieser Artikel soll helfen, sich im Dickicht dieser wichtigen Thematik zurechtzufinden, und konkrete Orientierungshilfen für sicheres Programmieren in der Java SE mit auf den Weg geben.

Java mit seiner inzwischen 26-jährigen Geschichte gilt gemeinhin als robuste und relativ sichere Programmiersprache. Klar gaben einzelne Vorfälle in der Vergangenheit, allen voran *Java Applets*, Anlass zu Bedenken. Applets sind heute aber auch von der Bildfläche verschwunden. Zur Ehrenrettung jener Java-Plug-ins sei bemerkt, dass es sich dazumal um eine Möglichkeit handelte, praktisch „beliebigen Code“ im Browser auszuführen. Mit anderen Worten, einem Programmierer standen mit Java als Universalsprache viel mehr Möglichkeiten offen als bei anderen Plug-ins mit spezifisch-beschränktem Funktionsumfang, zum Beispiel *Adobe Flash* oder auch *JavaScript*. Für das, was es konnte, stand es sicherheitsmäßig immer noch verhältnismäßig gut da [Ull20].

Der Vergleich ist allerdings unfair, wenn allein aufgrund der *Möglichkeit*, unbekannten, selbstständig nachladenden und nicht vertrauenswürdigen Code auszuführen, direkt auf die Sicherheit oder Unsicherheit einer Programmiersprache geschlossen wird. Wir mögen uns das Gedankenexperiment gar nicht ausmalen, wie es beispielsweise um die Sicherheit von „C/C++-Applets“ bestellt wäre. Dass Sicherheit auch nicht alleine an einer Sprache festgemacht werden kann, zeigt sich an den vielfältigen Maßnahmen, mit denen heute Geräte, Betriebssysteme und Applikationen abgesichert werden: Hardwareschutz, *System Integrity Protection* (bei macOS), zentrale App-Stores, signierte Applikationen, Sandboxen, Virtualisierung usw.

Sicherheit durch die virtuelle Maschine

Heute, im Jahre 2021, befindet sich Java im Programmiersprachen-Ranking immer noch unter den Top 3 – und das bereits seit über 20 Jahren [TIO]! Java hat vor allem im Middlewarebereich seinen festen Platz, aber auch große Desktop-Applikationen werden aufgrund ihrer Plattformunabhängigkeit gerne in Java realisiert.

Christian Heitzmann ist Java- und Python-zertifizierter Softwareentwickler mit einem CAS in Machine Learning und Inhaber der SimplexaCode AG in Luzern. Er entwickelt seit über 20 Jahren Software und unterrichtet bzw. doziert seit 10 Jahren unter anderem im Bereich der Java- und Python-Programmierung, Mathematik und Algorithmik.
E-Mail: christian.heitzmann@simplexacode.ch

Ein entscheidender Faktor für Javas Erfolg war und ist auf jeden Fall die *Java Virtual Machine (JVM)* als zusätzliche Abstraktionsschicht; eine Idee, die sich bis heute sehr bewährt hat und auch außerhalb der Java-Welt mehr und mehr Anklang findet [Sta20]. Genau dieser virtuellen Maschine ist es auch zu verdanken, dass Java-Programme vor einer Reihe potenzieller Programmierfehler und Sicherheitsschwachstellen auf niedrigen Ebenen verschont bleiben [WikiSecurity, Ull20]:

- Die JVM überprüft den Bytecode bereits vor seiner Ausführung, zum Beispiel auf unzulässige Verzweigungen, die Datensegmente als Befehle interpretieren könnten.
- Java unterstützt keine Pointer-Arithmetik.
- Array-Indizes werden zur Laufzeit stets auf ihre Gültigkeit überprüft.
- Typumwandlungen zwischen inkompatiblen Typen werden direkt unterbunden – wenn nicht zur Kompilierzeit, dann spätestens zur Laufzeit.
- Manuelle Allokation und Deallokation von Speicherbereichen ist weder nötig noch möglich.
- Ein *Garbage Collector* kümmert sich um die automatische Speicherbereinigung nicht mehr verwendeter Objekte.
- Ein *Security Manager* ermöglicht die Vergabe von Berechtigungen, zum Beispiel für den Zugriff auf das Dateisystem, die Verwendung von Netzwerkschnittstellen oder den Einsatz von Reflection innerhalb der JVM.

All diese Maßnahmen liefern einen großen Beitrag zum Speicherschutz, insbesondere vor den gefürchteten *Pufferüberläufen* (engl. *buffer overflows*) und *Stacküberläufen* (engl. *stack overflows*). Derartige Schutzmaßnahmen fehlen in anderen, maschinennahen Sprachen wie zum Beispiel C und C++ praktisch komplett. Sie sind deswegen auch deutlich anfälliger für derartige Fehler und lassen sich nur durch eiserne Disziplin beim Programmieren verhindern.

Sicherheitslücken in Java

Aber auch die an sich robuste Java-Sprachumgebung ist nicht vor Sicherheitslücken gefeit. Diese manifestieren sich zwar weniger bis gar nicht an „Bugs" des JDK oder der JVM selbst, dafür vermehrt an eigenen Programmierfehlern, die es potenziellen Angreifern ermöglichen, das Programmverhalten in nicht vorgesehener Art und Weise zu verändern.

Ausführungen zu sämtlichen sicherheitsrelevanten Bad Practices in Java könnten problemlos ein gesamtes Heft füllen. Wer bei der Recherche häufig eingesetzte Frameworks und Bibliotheken miteinbezieht, steht letztendlich gefühlt vor einem Fass ohne Boden. Im Rahmen dieses Artikels möchte ich daher eine möglichst kompakte Übersicht geben, muss mich dabei aber auf die *Java Platform, Standard Edition (Java SE)* beschränken. Die folgenden drei Fragen sollen dabei im Zentrum stehen:

- Von welchen Risiken müssen Java-Programmierer ausgehen?
- Wie manifestieren sich typische Schwachstellen im Code?
- Wo finden Java-Entwickler weitere Literatur zum Thema?

Benutzer und Entwickler als Sicherheitsrisiken

Das bekannte Bild des bösen, maskierten Hackers, der vor seinem Laptop sitzt, ist nicht nur völlig weltfremd, sondern trägt auch nichts zu einem realistischen Verständnis potenzieller Sicherheitslücken und einer sachgerechten Diskussion darüber bei. Wer als Bedrohung nur dieses eine Bild des Hackers im Kopf hat, kann sich der Thematik „Security" mit nur einem Totschlagargument entledigen: *„Mir will schon keiner was Böses!"*

[Nyg18] bringt die Gefahren, die von Endbenutzern ausgehen können, auf den Punkt:

- **Benutzer verbrauchen Ressourcen.** Je mehr Benutzer zum Beispiel einen Webservice nutzen, desto größer wird der Speicherdruck auf dem Server. Nicht wenige Programmfehler entstehen dadurch, dass sie in Sachen Speicherbedarf nicht vorausschauend geplant werden oder zu gutmütiges Caching betreiben, aber dann im Ernstfall träge reagieren, abstürzen oder aufgrund von Time-outs andere Schwachstellen zum Vorschein bringen. Im Besonderen können plötzliche und unerwartet hohe Besucherströme zu Abstürzen, Speicherplatzmangel, Deadlocks oder Race Conditions führen.
- **Benutzer tun seltsame und zufällige Dinge.** Sie werden sich weder vorhersehbar noch nachvollziehbar verhalten. Existiert ein Programmfehler, werden sie mit Sicherheit auf ihn stoßen. Es ist also unabdingbar, vorab nicht nur die „Happy Cases", sondern auch so viele Spezialfälle wie möglich (automatisiert) zu testen.
- **Es gibt sie, die böswilligen Benutzer.** Sie werden alles tun, um Schwachstellen im System zu finden und auszunutzen. Und das unabhängig davon, ob man an sie „glaubt" oder nicht. Oft haben sie auch etwas, was Entwickler nicht haben: Zeit.

Die eigentlichen Gefahren gehen also größtenteils von ganz normalen Endbenutzern ohne böse Absichten aus. Wie wir weiter unten noch sehen werden, gilt es aber nicht nur die Gruppe der Endbenutzer, sondern auch jene der Entwickler zu berücksichtigen. Letztere werden die Software entweder weiterentwickeln oder sie zumindest in Form von Bibliotheken benutzen. Code ist daher so zu schreiben, dass andere Entwickler weder absichtlich noch unabsichtlich potenzielle Schwachstellen ausnutzen können.

Im Buch „Code Craft" [Goo06] findet sich ein sehr guter allgemeiner, sprachenunabhängiger Überblick zu möglichen Sicherheitsrisiken in der Softwareentwicklung mitsamt den typischen Gründen und Ausreden im Entwicklungsprozess. Anschließend zeigt es Lösungswege, Richtlinien und Denkanregungen für die künftige Design- und Entwicklungsarbeit auf. Auch außerhalb der Security-Thematik ist das Buch in meinen Augen eine absolute Pflichtlektüre. Es wird der Clean-Code-Thematik wesentlich fundierter und differenzierter gerecht als der allgemein bekannte Bestseller von „Uncle Bob" (dessen Buch „Clean Code" das Thema Security vollständig ausklammert).

Als Grundproblem führt [Goo06] aus, dass es als Softwareentwickler viele Anforderungen (Funktionalität, Bedienbarkeit, Verlässlichkeit usw.) in einem zu engen Zeit- und Geld-Korsett umzusetzen gilt. Security wird deswegen oft als Nebensächlichkeit betrachtet. Im Fazit zeige ich Ihnen, wie Entwickler in einer Softwarefirma dieser spezialisierten Sicherheitsproblematik – trotz beschränkter Ressourcen – doch Herr werden können.

Kategorien potenzieller Sicherheitslücken in der Java SE

Wenn es konkret um die Sprache Java geht, dann sind Oracles „Secure Coding Guidelines for Java SE" [OraGuidelines] die Referenz schlechthin. Insgesamt 72 Richtlinien, aufgeteilt in 10 Themenbereiche, liefern wertvolle Hinweise und ein Bewusstsein für sichereres Programmieren in der Java SE. Als Lesedauer sollten mindestens zwei Stunden veranschlagt werden. Ich denke hingegen, die Inhalte sind „zu harte Kost", um sie in einem Rutsch „einfach mal durchzulesen". Wieso nicht einfach mal per Zufallsgenerator einen Abschnitt auslosen und sich diesen dann konzentriert zu Gemüte führen, wie in meinem letzten Artikel vorgeschlagen [Hei21]?

Die erwähnten Guidelines geben eine gute Übersicht, in welchen Bereichen besonders auf Sicherheitsaspekte geachtet werden muss. Bereits der erste einleitende Absatz bringt es auf den Punkt: *„While the Java security architecture can in many cases help to protect users and systems from hostile or misbehaving code, it cannot defend against implementation bugs that occur in* trusted *code. Such bugs can inadvertently open the very holes that the security architecture was designed to contain."*

Es geht also darum, mit „schlechtem Code" nicht unfreiwillig Löcher in die per se sichere Architektur zu bohren. Dazu zählt vor allem, die üblichen Prinzipien und **Best Practices der modernen objektorientierten Programmierung** zu beherzigen:

- Vermeidung von Code-Duplikationen,
- Kapselung,
- saubere Auftrennung von Schnittstellen,
- Überprüfung von Argumenten,
- Fail-fast-Ansätze,
- Minimieren von Sichtbarkeiten,
- Unterbinden unabsichtlicher Klassenerweiterungen,
- Bevorzugung unveränderlicher (immutable) statt veränderlicher (mutable) Klassen,
- Erzeugung defensiver Kopien sowohl bei Eingaben als auch bei Rückgaben,
- Beschränkung von Konstruktoren.

Eine weitere wichtige Kategorie bildet die **vorgängige Prüfung von Eingaben** in Form von Strings, URLs, Dateien oder gar Klassen:

- Bestimmte Dateiformate können bei präparierten, ungeprüften Eingaben unverhältnismäßig viele Ressourcen in Anspruch nehmen und so Dienste lahmlegen, zum Beispiel als Pixelgrafiken zu rendernde SVG-Dateien, BMP-Dateien im Allgemeinen, ZIP-Bomben, XML-Entitäten, Hash-Tabellen, reguläre Ausdrücke und viele andere.
- Texteingaben können – wenn sie nicht auf Sonderzeichen überprüft und escapet werden – in den darauffolgenden Stufen versehentlich zur Interpretation gelangen, zum Beispiel in URLs, als Teilstrings in dynamischem SQL oder als Teil von Pfadangaben (`"../"`).
- Die Serialisierung und Deserialisierung nicht vertrauenswürdiger Daten via `ObjectInputStream` und `ObjectOutputStream` ist hochgradig unsicher und sollte grundsätzlich vermieden werden. Stattdessen soll auf heute etablierte Serialisierungsformate wie zum Beispiel *JSON* und deren namhafte Bibliotheken gesetzt werden [BloItem85].
- Übermäßig viele Aufrufe innerhalb kurzer Zeit können zu einem *Denial of Service (DoS)* führen. Dabei können Logger den Speicherplatz mit ihren (zu detaillierten) Logeinträgen füllen.

Der letzte Punkt, das Logging, bildet auch die Überleitung zu einer dritten wichtigen Kategorie potenzieller Fehlerquellen, nämlich die **unfreiwillige Preisgabe sensitiver Informationen**:

- Logdateien dürfen (nur schon aus rechtlichen Gründen) keine sensiblen Daten wie zum Beispiel Personendaten, Versicherungsnummern oder Passwörter enthalten.
- Hochsensible Daten sollten nach ihrer Verwendung sogar direkt aus dem Arbeitsspeicher gelöscht werden; siehe hierzu auch die „Security note" in der Klassenbeschreibung von `Console` und seiner `readPassword`-Methoden [JAPIConsole].
- Exceptions und deren Stack-Traces – unabhängig davon, ob sie geloggt oder auf der Konsole ausgegeben werden – können auch Daten preisgeben, die von den Entwicklern so nicht angedacht waren, zum Beispiel lokale Dateipfade, Variableninhalte oder Methodenaufrufketten. Aus diesen lässt sich relativ einfach auf Interna einer Applikation oder eines Webservice schließen, was wiederum Türen für weitere Angriffsziele öffnet. Eine Bereinigung von Exception-Nachrichten kann hierbei helfen, ebenso die in meinem letzten Artikel präsentierte Idee, komplett auf Exception-Texte zu verzichten und stattdessen einfach einen hartcodierten *Zufallsbezeichner* einzusetzen [Hei21]. Der Vollständigkeit halber sei auch erwähnt, dass mit Java 15 das Flag `ShowCodeDetailsInExceptionMessages` neu auf `true` gesetzt wurde [OraNotes]. Damit sind die detaillierten Exception-Messages aus JEP 358 neu standardmäßig aktiv und zeigen sich im Falle einer `NullPointerException` sehr gesprächig, was wahrscheinlich nicht in jedermanns Interesse ist [JEP].

Statische Code-Analyse-Tools

Auch die statischen Analyse-Tools von *SonarSource* (und wahrscheinlich auch von anderen Herstellern) liefern eine gute Übersicht potenzieller Sicherheitsschwachstellen im Code. Nebst ihrer eigentlichen Aufgabe, bei sicherheitskritischen Stellen sofort eine Warnung auszulösen, verfügt die Tool-Palette um *SonarLint*, *SonarCloud* und *SonarQube* über eine hervorragende Dokumentation sämtlicher von ihnen überwachter Regeln, inklusive konformer und nicht konformer Code-Beispiele. Für Java führt das Regelwerk aktuell 43 *Vulnerabilities* und 29 *Security Hotspots* auf, doch es gibt auch Regeln für viele andere Programmiersprachen [SonVulner, SonSecurity].

Es lohnt sich auf jeden Fall, ein wenig darin zu stöbern. Ein kleiner Auszug aus dem Regelwerk beinhaltet zum Beispiel:

- Hartcodieren von Anmeldeinformationen, inkl. IP-Adressen.
- Entpacken von Archiven ohne Kontrolle des Ressourcenverbrauchs oder der Namen der Dateieinträge.
- Einsetzen schwacher Hash-Algorithmen oder Pseudozufallszahlengeneratoren.
- Vorhersagbares Initialisieren von `SecureRandom`.
- Ausliefern von Produktivcode mit eingeschalteten Debug-Features.
- Dynamisches Laden von Klassen.

Effective Java

Joshua Bloch führt in seinem Standardwerk „Effective Java" über das ganze Buch verteilt immer wieder Sicherheitsprobleme mit sehr großem Detaillierungsgrad auf. Es gehört nicht nur deswegen zur Pflichtlektüre eines jeden professionellen Softwareentwicklers. Ein paar seiner Ausführungen hier in der Kurzfassung:

- Es gibt mehrere Arten, **Singletons** zu implementieren. Via Reflection lassen sich auch private Konstruktoren aufrufen und damit mehrere Instanzen erzeugen; ähnlich verhält es sich bei der Deserialisierung solcher Klassen. *Enums* stellen die sicherste Variante zur Implementierung von Singletons dar [BloItem3]. Für weitere Hintergründe zu Enums, inklusive ihrer Einsätze als Singleton, darf ich auf meinen JavaSPEKTRUM-Artikel von 2019 verweisen [Hei19].
- **Finalizer-Angriffe** fangen als Unterklasse eine Exception des Konstruktors ihrer Oberklasse ab, speichern eine Referenz auf die „halb erzeugte" Klasse und können anschließend mit ihr machen, was immer sie wollen. Verhindern lässt sich dies durch das Unterbinden von Subklassen (`final class`) oder einer finalen, leeren `finalize`-Methode in der Oberklasse (`protected final void finalize() {}`) [BloItem8].
- Allgemein sollte man bei Klassen **Komposition der Vererbung vorziehen**. Werden zum Beispiel Wrapper-Klassen via Vererbung realisiert, und kommt in einer neuen Version der Oberklasse eine neue Methode hinzu, so entsteht in der Unterklasse dort ein Loch, wo die Unterklasse diese neue Methode noch nicht überschrieben hat. Auf diese Art können die notwendigen Bedingungsprüfungen des Wrappers umgangen werden [BloItem18]. Wird der Wrapper hingegen via Komposition realisiert, dann steht die neue Methode für den Aufrufer einfach (noch) nicht zur Verfügung.
- Der **Time-of-Check/Time-of-Use-Angriff (TOCTOU)** macht sich die Schwachstelle zunutze, dass sich ein veränderliches Objekt (zum Beispiel das alte `java.util.Date`) als Konstruktor- oder Methodenargument zwischen dem Zeitpunkt seiner Überprüfung (zum Beispiel, ob das Datum ≥ 1.1.2021 ist) und dem Zeitpunkt seines Einsatzes (zum Beispiel dem Abspeichern in einer Datenbank) inhaltlich ändern kann. Vermeiden lässt sich dieses Problem mit einer defensiven Kopie dieses Eingabeparameters oder – sofern das machbar ist – einer Implementierung der Klasse des Parameters als unveränderliche Klasse [BloItem50].

Fazit

Wir haben gesehen, dass wir uns in Java wenig Sorgen über „Low-Level-Sicherheitslücken" machen müssen. Vielmehr können eigene Nachlässigkeiten während des Entwicklungsprozesses „auf hoher Ebene" schnell sicherheitsrelevante Schwachstellen einführen. Kenntnisse um die verschiedenen Bereiche, in denen solche Probleme auftreten können, sind schon die halbe Miete. Als potenzielle Risiken sollen vor allem künftige Endanwender und Softwareentwickler berücksichtigt werden. Software und Code sind so abzusichern, dass sie sowohl Falschanwendungen (ohne böse Hintergedanken) als auch aktiver Sabotage standhalten. Ich hoffe, dieser Artikel konnte einen entscheidenden Beitrag dazu leisten, sich mit der Sicherheitsthematik und seiner einschlägigen Literatur vertiefter auseinanderzusetzen. Die besonders hilfreichen Quellen sind in nachfolgendem Literaturverzeichnis **fett** gedruckt.

Es ist verständlich, wenn sich nicht eine ganze Entwicklungsabteilung mit stundenlangen Studien zu dieser Thematik befassen kann, und anschließend auch noch der Anspruch besteht, dass alle Entwickler sämtliche Inhalte stets geistig präsent haben. Vielleicht wäre es eine Option, ein oder zwei hierfür speziell ausgebildete Entwickler im Haus zu haben, die sämtliche Pull-Requests zusätzlich einem obligatorischen Sicherheits-Review unterziehen?

Literatur und Links

[BloItem3/8/18/50/85] J. Bloch, Effective Java, Items 3/8/18/50/85, 3rd Edition, Addison-Wesley, 2018

[Goo06] P. Goodliffe, Code Craft: The Practice of Writing Excellent Code, Chapter 12: An Insecurity Complex, No Starch Press, 2006

[Hei19] C. Heitzmann, Der Einsatz von enums abseits reiner Aufzählungen, in: JavaSPEKTRUM, 1/2019

[Hei21] C. Heitzmann, Der Einsatz von Zufallsbezeichnern in der Softwareentwicklung und im Alltag, in: JavaSPEKTRUM, 4/2021

[JAPIConsole] Java Platform, Standard Edition & Java Development Kit Version 16 API Specification, Class Console, https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/io/Console.html

[JEP] OpenJDK, JEP 358: Helpful NullPointerExceptions, https://openjdk.java.net/jeps/358

[Nyg18] M. T. Nygard, Release It!, Chapter 4: Stability Antipatterns, Users, 2nd Edition, Pragramatic Bookshelf, 2018

[OraNotes] Oracle, JDK 15 Release Notes, Enable ShowCodeDetailsInExceptionMessages by default, https://www.oracle.com/java/technologies/javase/15-relnote-issues.html#JDK-8233014

[OraGuidelines] Oracle, Secure Coding Guidelines for Java SE, Version 8.0, https://www.oracle.com/java/technologies/javase/seccodeguide.html

[SonSecurity] SonarSource, Java static code analysis, Security Hotspot, https://rules.sonarsource.com/java/type/Security%20Hotspot

[SonVulner] SonarSource, Java tatic code analysis, Vulnerability, https://rules.sonarsource.com/java/type/Vulnerability

[Sta20] M. Stal, Java rocks! – Eine Liebeserklärung, in: JavaSPEKTRUM, 4/2020

[TIO] TIOBE Index, https://www.tiobe.com/tiobe-index/

[Ull20] C. Ullenboom, Java ist auch eine Insel, Abschnitt 1.2: Warum Java populär ist – die zentralen Eigenschaften, 15. Auflage, Rheinwerk Verlag, 2020

[WikiSecurity] Wikipedia, Security of the Java software platform, https://en.wikipedia.org/wiki/Security_of_the_Java_software_platform

IT-Tage 2021
REMOTE KONFERENZ
06.12.-09.12.2021
Konferenz für Entwicklung, Architektur, KI, Datenbanken, DevOps und Agile
Ihre Vorteile
+ 18 Subkonferenzen
+ Mehr als 200 Vorträge
+ Vortrags-Aufzeichnungen
+ Umfassende Konferenz-Plattform
Jetzt anmelden!
www.it-tage.org