www.javaspektrum.de

Ausgabe 1/2019
D € 9,90 · A € 10,90 · CHF 18,20

4 194156 109901 01

JavaSPEKTRUM

Magazin für professionelle Entwicklung und Integration von Enterprise-Systemen

Bewölkt und sonnig – Programmierung in der Cloud

Interview

Interview mit dem KI-Spezialisten Chris Boos darüber, welche Ausmaße Künstliche Intelligenz in zehn Jahren annimmt

Fachthemen

DevOps und modellbasiertes Testen als Effizienz-Turbo

Der Einsatz von enums abseits reiner Aufzählungen

zum Inhalt

Sonderbeilage: German Testing Magazin „Agiles Testen“

Bewölkt und sonnig – Programmierung in der Cloud

G30325

Vom Mauerblümchen zur Wunderpille

Der Einsatz von enums abseits reiner Aufzählungen

Christian Heitzmann

Wer `enum`-Aufzählungen nur als eine bequeme und typsichere Auflistung von Werten verwendet, unterschätzt ihr Potenzial. Sie sind nämlich vollwertige Java-Klassen, die sich zum einen mit weiterer Funktionalität „pimpen“ lassen, zum anderen als elegante Abkürzung für Klassen mit gewissen Eigenschaften dienen. Versteht man erst die Korrespondenz zwischen der `enum`-Notation und der „auskompilierten“ Klasse, so wird man sie künftig wesentlich routinierter einsetzen können.

Bis und mit Java 1.4 war es üblich, Aufzählungen von Werten mithilfe von konstanten Integer-Werten zu realisieren, zum Beispiel:

```
public static final int RED = 1; [...] GREEN = 2; [...] BLUE = 3;
```

Die Probleme eines solchen „Pattern“ lagen auf der Hand: keinerlei Typsicherheit, keinerlei Schutz vor unzulässigen Werten oder vertauschten Methodenparametern, dafür die Möglichkeit sinnloser Operationen (zum Beispiel das Multiplizieren von Konstantenwerten), um nur ein paar davon zu nennen. Joshua Bloch stellte dazumal in seinem berühmten Buch *Effective Java* ein *Typesafe Enum Pattern* [Bloch21] vor, welches den Großteil erwähnter Probleme zwar löste, aber sehr viel manuelle Codierarbeit nach sich zog und somit nicht wirklich praxistauglich war.

Ab Java 5 bot die Sprache mit dem `enum`-Befehl endlich ein bequemes Sprachmerkmal an, welches im Wesentlichen Joshua Blochs Entwurfsmuster unter der Haube implementierte. Interessanterweise hat man es mit dem `enum`-Schlüsselwort aber nicht dabei belassen, sondern derartige Aufzählungen erlauben noch viel mehr, als sie auf den ersten Blick vermuten lassen.

Einfache enum-Aufzählung

Listing 1 zeigt eine klassische Definition und Verwendung einer `enum`-Aufzählung. Eine `Color` kann genau die vier Werte `CYAN`, `MAGENTA`, `YELLOW` und `BLACK` annehmen. Soll eine solche Farbe als Methodenparameter verwendet werden, so wird sie ganz einfach in der Methodensignatur aufgeführt. Der Aufruf erfolgt selbsterklärend zum Beispiel über `checkAndPrintColor(Color.CYAN)`.

```
public final class PlainEnumDemo {
  private enum Color {
    CYAN, MAGENTA, YELLOW, BLACK
  }

  public static void main(String[] args) {
    PlainEnumDemo demo = new PlainEnumDemo();
    demo.checkAndPrintColor(Color.CYAN);
  }
  private void checkAndPrintColor(Color color) {
    System.out.println(color);
    if (color == Color.BLACK) {
      System.out.println("Key color");
    }
  }
}
```

Listing 1: Einfache `enum`-Aufzählung

Der Compiler stellt sicher, dass keine ungültigen, undefinierten Farben ins Spiel gebracht werden, und garantiert ebenso zuverlässig, dass die Methode `checkAndPrintColor(Color)` zum Beispiel nicht mit einer `Weekday`-Aufzählung oder einem der 12 Sternzeichen aufgerufen wird. All dies wäre bei den konstanten Integer-Werten von frü-

Christian Heitzmann ist Gründer und Geschäftsführer der SimplexaCode AG in Luzern, die sich auf Softwareentwicklung, -schulung und -beratung vor allem für MINT-Anwendungen und technische Implementierungsthemen in Java spezialisiert hat. Er ist seit 15 Jahren mit Java vertraut und hat während vieler Jahre Algorithmen und Mathematik unterrichtet.
E-Mail: christian.heitzmann@simplexacode.ch

her möglich gewesen. Es war wahrlich kein einfaches Unterfangen, ohne Compilerschutz derartige Fehler später aufzuspüren und zu beheben.

Ebenfalls sehr elegant ist die Überprüfung von `enum`-Werten innerhalb eines `if`-Konstruktes. Man könnte vermuten, dass dort die typische Java-Falle betreffend Referenzvergleich via `==` und Wertevergleich via `equals` zuschnappt, aber nein, ein einfaches `==` reicht hier vollkommen aus.

Wieso das so ist, und wie und warum sich die `enum`-Aufzählungen auch noch so bequem erweitern lassen, versteht man am besten, wenn man sich einmal genau anschaut, was der Compiler im Hintergrund mit dem `enum`-Befehl macht. Im Wesentlichen handelt es sich beim `enum`-Schlüsselwort nämlich nur um „syntactic sugar", also um ein Sprachmerkmal, das streng genommen überhaupt keine neue Funktionalität anbietet, sondern einfach nur dem Programmierer die Arbeit „versüßt". Fast alles, was `enum` kann, wäre auch mit dem Java 1.4-Befehlssatz möglich gewesen.

Zum Dekompilieren der Java-Klasse verwendete ich den Decompiler *CFR* [CFR] und konnte dabei auf die schnelle und hilfsbereite Unterstützung des Entwicklers zählen. Aus *Eclipse* heraus kompilierte Klassen weisen nämlich ein anderes Format auf als die des „offiziellen" JDK-Compilers (`javac`), was vereinzelt zu inkonsistenten Ergebnissen beim Dekompilieren führte, die aber mittlerweile behoben sind.

Selbstverständlich erkennt ein Java-Decompiler, wenn er auf Bytecode stößt, der aus `enum`s resultiert, und wandelt ihn entsprechend auch wieder in `enum`s zurück um. Um genau das zu verhindern (wir möchten ja sehen, wie `enum` aussähe, wenn es kein `enum` gäbe), ruft man CFR mit der Option `--sugarenums false` auf. So wird das syntaktische „Versüßen" beim Dekompilieren unterbunden. Listing 2 zeigt die (in der Formatierung leicht angepasste) Ausgabe nach dem (ungezuckerten) Dekompilieren mit CFR.

Ein paar Besonderheiten fallen auf:

- Jede `enum`-Aufzählung (hier: `Color`) ist automatisch *statisch*, auch wenn man bei ihrer Definition auf das `static`-Schlüsselwort verzichtet. Wäre `Color` im hier vorliegenden Beispiel `public` und nicht `private`, wäre ein Referenzieren des Aufzählungstyps von einer anderen Klasse heraus via `PlainEnumDemo.Color` problemlos möglich, es bräuchte also nicht zwingend erst eine Instanz von `PlainEnumDemo`.
- Jede `enum`-Aufzählung ist automatisch *unveränderbar*, da sie vom Compiler (unter anderem) mit dem `final`-Schlüsselwort versehen wird. Das heißt zum einen, dass man `enum`s nicht aus Versehen „kaputt machen", überschreiben oder löschen kann, zum anderen aber auch, dass man `enum`-Aufzählungen nicht voneinander erben lassen kann. Ein sinngemäßes `enum MoreColors extends Colors { ORANGE, VIOLET, BROWN }` ist also bewusst *nicht* möglich.
- Jede `enum`-Aufzählung vom Typ `E` *erweitert die abstrakte Basisklasse* `java.lang.Enum<E>`. [JAPIEnum] Diese stellt als Oberklasse aller `enum`s ein paar Methoden bereit, die aber für die Praxis nicht sehr relevant sind. Erwähnenswert ist allerdings, dass `Enum<E>` die Schnittstellen `Comparable<E>` und `Serializable` implementiert, und somit jede `enum`-Aufzählung auch sortierbar und serialisierbar ist.
- Aufpassen muss man beim Versuch, `Enum<E>` selbst implementieren zu wollen. Dies wird der Compiler nämlich verbieten. Wer versucht, die hier abgedruckte dekompilierte Klasse wiederum in seine IDE als Quelltext einzufügen, wird vom Compiler die Fehlermeldung *„The type Color may not subclass Enum<PlainEnumDemo.Color> explicitly"* erhalten. Die *Java Language Specification* hebt diese Besonderheit auch ausdrücklich hervor: *„It is a compile-time error to attempt to explicitly instantiate an enum type."* [JLSEnum]
- Jeder Wert (hier also `CYAN`, `MAGENTA`, `YELLOW` und `BLACK`) eines Aufzählungstyps (hier: `Color`) *ist vom Typ der Aufzählung selbst*. `Color.CYAN` ist also vom Typ `Color`. So wird (im Gegensatz zu den eingangs erwähnten Integern) die Typsicherheit bei Aufzählungen gewährleistet.
- Jede `enum`-Aufzählung enthält einen *privaten Konstruktor*, lässt sich also nie „von außen" erzeugen. Zusammen mit der oben erwähnten `final`-Eigenschaft und der Tatsache, dass der Compiler jeglichen Versuch, `Enum<E>` manuell zu erweitern, gemäß Spezifikation unterbindet, ist also garantiert, dass jeder `enum`-Wert zur Laufzeit nur *ein einziges Mal* vorkommt. Jede `enum`-Aufzählung stellt damit einen *Singleton* dar. Überprüfungen auf Gleichheit mittels `==` sind legitim, da es sich bei zwei gleichen `enum`-Werten garantiert auch um die gleichen Referenzen handelt.

Zusammenfassend kann festgehalten werden, dass vom Compiler aus `enum` eine Singleton-Klasse generiert wird, die darüber hinaus komparabel und serialisierbar ist. Dass sich einer solchen Klasse eigene Methoden und Attribute hinzufügen lassen müssten, liegt eigentlich auf der Hand.

```
public final class PlainEnumDemo {
  private static final class Color extends Enum<Color> {
    public static final /* enum */ Color CYAN
      = new Color("CYAN", 0);
    public static final /* enum */ Color MAGENTA
      = new Color("MAGENTA", 1);
    public static final /* enum */ Color YELLOW
      = new Color("YELLOW", 2);
    public static final /* enum */ Color BLACK
      = new Color("BLACK", 3);
    private static final /* synthetic */ Color[] $VALUES;

    static {
      $VALUES = new Color[] { CYAN, MAGENTA, YELLOW, BLACK };
    }
    private Color(String string, int n) {
      super(string, n);
    }

    public static Color[] values() {
      return (Color[]) $VALUES.clone();
    }
    public static Color valueOf(String string) {
      return Enum.valueOf(Color.class, string);
    }
  }
  /* [...] Methods omitted. */
}
```

Listing 2: Dekompilierte Klasse aus Listing 1

enum-Aufzählung mit statischen Methoden

Listing 3 fügt unserer eigenen `Color`-`enum`-Aufzählung eine statische Methode `isAchromatic(Color)` hinzu. Sie gibt bei einer unbunten Far-

be, die ihr übergeben wird, `true` zurück. Des Weiteren zeigt ihr Quelltext (ohne weitere Bemerkungen), wie einfach und elegant sich `enum`-Aufzählungen innerhalb von `switch`-Kontrollstrukturen verwenden lassen.

```
public final class EnumWithStaticMethodDemo {
  private enum Color {
    CYAN, MAGENTA, YELLOW, BLACK; /* Note the semicolon! */

    public static boolean isAchromatic(Color color) {
      switch (color) {
      case BLACK:
        return true;
      case CYAN: case MAGENTA: case YELLOW:
        return false;
      default:
        throw new AssertionError();
      }
    }
  }

  public static void main(String[] args) {
    EnumWithStaticMethodDemo demo = new EnumWithStaticMethodDemo();
    demo.checkAndPrintColor(Color.MAGENTA);
  }
  private void checkAndPrintColor(Color color) {
    System.out.println(color);
    if (Color.isAchromatic(color)) {
      System.out.println("Achromatic");
    }
  }
}
```

Listing 3: `enum`-Aufzählung mit statischer Methode

Wichtig ist das Semikolon nach der Definition der Aufzählungswerte:

```
CYAN, MAGENTA, YELLOW, BLACK;
```

Der Strichpunkt ist immer dann notwendig, wenn den Aufzählungswerten noch andere (eigene) Attribute und Methoden folgen.

enum-Aufzählung mit Attributen und Methoden

Jede Definition eines Aufzählungswertes innerhalb von `enum` ist nichts anderes als eine abgekürzte Schreibweise für ein `public-static-final`-Attribut in der generierten Klasse. Der Bezeichner `MAGENTA` zum Beispiel führt zu einer Deklaration des entsprechenden gleichnamigen Attributes inklusive eines initialisierenden Konstruktoraufrufs:

```
public static final Color MAGENTA = new Color("MAGENTA", 1);
```

Möchte man nicht den Standardkonstruktor mit den Parametern `(String name, int ordinal)` verwenden, so kann man einen eigenen (privaten!) Konstruktor definieren. Listing 4 zeigt ein solches Beispiel auf.

Der `enum`-Aufzählung wurden nun ein eigenes String-Attribut `htmlColorCode`, ein eigener Konstruktor `Color(String)` und eine eigene, ganz normale Instanzmethode `getHtmlColorCode` hinzugefügt. Um den eigenen Konstruktor auch aufrufen zu können, fügt man hinter der Deklaration eines Aufzählungswertes einfach die entsprechende Parameterliste an, zum Beispiel `YELLOW(„#FFFF00“)`.

Auf diese Art lassen sich den einzelnen Aufzählungswerten zusätzliche Informationen mitgeben. Im aufgezeigten Beispiel in Listing 4 enthält zum Beispiel jede einzelne Farbe zusätzlich noch die Information über ihren HTML-Farbcode.

```
import java.util.*;

public final class EnumWithFieldAndMethodDemo {
  private enum Color {
    CYAN ("#00FFFF"),
    MAGENTA("#FF00FF"),
    YELLOW ("#FFFF00"),
    BLACK ("#000000");

    private String htmlColorCode;

    private Color(String htmlColorCode) {
      Objects.requireNonNull(htmlColorCode);
      this.htmlColorCode = htmlColorCode;
    }

    public String getHtmlColorCode() {
      return htmlColorCode;
    }
  }

  public static void main(String[] args) {
    EnumWithFieldAndMethodDemo demo = new EnumWithFieldAndMethodDemo();
    demo.printColor(Color.YELLOW);
  }
  private void printColor(Color color) {
    System.out.println(color);
    System.out.println(color.getHtmlColorCode());
  }
}
```

Listing 4: `enum`-Aufzählung mit Attributen, Methoden und Konstruktoren

enum-Aufzählung mit eigenen Konstantennamen

Im ersten Beispiel (Listing 1) wurde bereits demonstriert, dass sich `enum`-Konstanten ganz einfach via `System.out.println` ausgeben lassen. Auch dies ist ein klarer Vorteil gegenüber der Realisierung von Aufzählungswerten mithilfe von Integern, vor allem zu Debugging- oder Loggingzwecken.

Der vermutlich häufigste Anwendungsfall für die Anpassung oder Erweiterung eines `enum`-Typs besteht wohl darin, eine eigene String-Repräsentation für die jeweiligen Aufzählungswerte zu hinterlegen. Dies lässt sich ganz einfach analog dem bereits in Listing 4 vorgestellten Schema realisieren.

```
import java.util.*;

public final class EnumWithOwnNamesDemo {
  private enum Color {
    CYAN ("Cyan"),
    MAGENTA("Magenta"),
    YELLOW ("Gelb"),
    BLACK ("Schwarz");

    private String name;

    private Color(String name) {
      Objects.requireNonNull(name);
      this.name = name;
    }

    @Override
    public String toString() {
      return name;
    }
  }
  /* [...] Methods omitted. */
}
```

Listing 5: `enum`-Aufzählung mit eigenen Konstantennamen

Listing 5 verwendet für die `enum`-Aufzählung wieder einen eigenen privaten Konstruktor, dem die gewünschte String-Repräsentation des Namens mitgegeben wird. In diesem Fall soll jede Farbe einen deutschen Namen erhalten. Zusätzlich wird die Methode `toString` überschrieben, welche dann den im (neuen) Attribut `name` gespeicherten String zurückgibt.

enum-Aufzählung mit überschreibenden toString-Methoden

Möchte man nicht alle, sondern nur einzelne `enum`-Werte mit eigenen Namen versehen, so bietet es sich an, für diese Werte direkt die `toString`-Methode zu überschreiben. Im Beispiel in Listing 6 werden keine deutschen Übersetzungen für `CYAN` und `MAGENTA` benötigt, da sie genau gleich wie auf Englisch lauten. Für `YELLOW` und `BLACK` hingegen werden jeweils direkt in einer inneren anonymen Klasse die Methoden `toString` überschrieben, sodass sie den entsprechenden deutschen Namen zurückgeben.

Selbstverständlich ist das Überschreiben auch für andere Methoden – nicht nur für `toString` – möglich. Interessant ist sicherlich die Möglichkeit, innerhalb der Definition eines `enum`-Aufzählungs*typs* eine (allenfalls abstrakte) Methode vorzugeben, die dann bei der Definition der einzelnen `enum`-Aufzählungs*werte* jeweils individuell überschrieben wird. Es ist sogar möglich, dass der eigene `enum`-Aufzählungstyp ein Interface implementiert. Lautet die Deklaration zum Beispiel `enum Color implements MyInterface`, so muss der `enum`-Typ oder *jeder* `enum`-Wert die Interface-Methoden implementieren.

Möchte man das Überschreiben (eher „Implementieren“) für jeden Wert erzwingen, so sollte die entsprechende Methode `abstract` deklariert sein oder ein Interface erweitern. Ansonsten ist das Überschreiben – wie im Beispiel mit `toString` – optional.

```
public final class EnumWithOverridenToStringMethodsDemo {
  private enum Color {
    CYAN,
    MAGENTA,
    YELLOW {
      @Override
      public String toString() {
        return "GELB";
      }
    },
    BLACK {
      @Override
      public String toString() {
        return "SCHWARZ";
      }
    }
  }
  /* [...] Methods omitted. */
}
```

Listing 6: `enum`-Aufzählung mit überschreibenden toString-Methoden

enum-Aufzählung als Singleton

Beim Untersuchen des dekompilierten Codes haben wir bereits gesehen, dass ein `enum` unter der Haube ein Singleton repräsentiert. Ein Singleton ist eine Klasse, von der jeweils nur *eine einzige* Instanz während der Laufzeit einer Applikation existiert.

Listing 7 zeigt die klassischen Maßnahmen, die nötig sind, um dies für eine normale Klasse zu realisieren.

```
import java.util.concurrent.*;

public final class ClassicSingleton {
  private static final ClassicSingleton INSTANCE
    = new ClassicSingleton();
  private int value
    = ThreadLocalRandom.current().nextInt();

  public static ClassicSingleton getInstance() {
    return INSTANCE;
  }
  public int getValue() {
    return value;
  }
}
```

Listing 7: Klassisches Singleton

- Eine Singleton-Klasse muss *final* sein, damit man sie nicht erweitern und somit den Singleton-Mechanismus in einer Unterklasse aushebeln kann.
- Die (einzige) Instanz einer Singleton-Klasse wird in einem *statischen Attribut* gespeichert – meist mit dem Namen `INSTANCE` oder `instance`.
- Der *Konstruktor* eines Singletons muss *privat* sein, damit sich von außen keine Instanzen erzeugen lassen.
- Der Zugriff auf einen Singleton erfolgt nur über eine *statische Zugriffsmethode* – meist mit dem Namen `getInstance`.

Im Code-Beispiel in Listing 7 wird im Singleton ein Zufallswert im Attribut `value` gespeichert, der über die Methode `getValue` abgerufen kann. Dieser Zufallswert wird nur bei der Erzeugung des Singletons gesetzt und soll demonstrieren, dass ein Singleton während der Laufzeit eines Programmes wirklich kein weiteres Mal erzeugt wird. Trotz mehrerer Aufrufe – egal in welcher Klasse, in welchem Paket oder in welchem Thread – wird `value` immer den gleichen Wert enthalten.

Da ein `enum` automatisch ein Singleton darstellt, lässt sich seine Definition nutzen, um auf einfachere Art einen solchen zu realisieren. Der entsprechende Quelltext ist zwar nicht weltbewegend kürzer, dafür wesentlich einfacher. Denn Hand aufs Herz: Wer kann schon die vier Schritte zur Realisierung eines Singletons sofort aus dem Ärmel schütteln, ohne im Internet spicken zu müssen?

```
import java.util.concurrent.*;

public enum EnumSingleton {
  INSTANCE;

  private int value
    = ThreadLocalRandom.current().nextInt();

  public int getValue() {
    return value;
  }
}
```

Listing 8: `enum`-Aufzählung als Singleton

Listing 8 zeigt die triviale Realisierung via `enum`. Mit der Deklaration des Wertes `INSTANCE` ist die Arbeit erledigt. Die anderen Attribute und Methoden im Listing (also `value` und `getValue()`) sind ja die „eigenen“, die nicht Teil der Singleton-Implementierung, sondern der eigenen Programmlogik sind.

Sichere (De-)Serialisierbarkeit von enum-Singletons

Die Bedeutung des Java-eigenen (De-)Serialisierungsmechanismus hat mit der Verbreitung von Datenformaten wie *JSON* [JSON] heute an Bedeutung verloren. Zu groß waren und sind die Sicherheitsrisiken, die mit dem Deserialisieren von Byte-Streams (im schlimmsten Fall unbekannter Herkunft) einhergingen. [JSCGSer] Dennoch darf man dieses Thema auch heute noch nicht außer Acht lassen. Legacy-Code kann zum Beispiel immer noch auf dem Serialisierungsmechanismus basieren, und es ist darum unabdingbar, sich über dessen Sicherheitsrisiko bewusst zu sein.

Eine eingehende Behandlung der Hintergründe würde den Rahmen dieses Artikels sprengen, aus diesem Grund hier nur die ganz kurze Quintessenz: Eines der Grundprobleme des Deserialisierungsmechanismus besteht darin, dass er Objekte unter Umgehung der Konstruktorenaufrufe erzeugen kann. Er kann praktisch direkt aus einem Byte-Stream Objekte „backen“ und im Speicher platzieren. Entwurfsmustern wie Singletons macht dies natürlich einen dicken Strich durch die Rechnung.

Zwar gäbe es Richtlinien, wie man die Erzeugung von Objekten „unter Umgehung des Dienstweges“ verhindern kann, namentlich, indem man ausnahmslos alle Attribute als `transient` markiert und eine eigene `readResolve`-Methode zur Verfügung stellt. Ein kleinster Programmierfehler reicht aber aus, um einen Angreifer im schlimmsten Fall präparierte Byte-Streams einschleusen zu lassen, die dann – trotz Singleton-Eigenschaft – eine parallel existierende Instanz des Singletons zur Folge haben.

Der Compilerschutz bei `enum`s verhindert von Haus aus solche Probleme. Es ist nicht möglich, ohne Verwendung von Reflection und der Umgehung von Zugriffsrechten einen parallel existierenden `enum`-Singleton zu erzeugen. Dies ist ein weiterer Grund, wieso Singletons nur mithilfe von `enum`s implementiert werden sollten.

Zusammenfassung

`enum`s sind eine Wunderpille in Java. Seit Version 5 lösen sie elegant die bis dahin verwendeten Integer-Konstantenpools ab. Sie erhöhen die Lesbarkeit sowohl des Codes als auch der Programmausgaben. Nebst verschiedenen Möglichkeiten, eigene String-Repräsentationen zu hinterlegen, stellen `enum`s vollwertige Klassen dar, die sich mit eigenen Konstruktoren, Attributen, statischen und nicht statischen Methoden versehen lassen. Ihre impliziten Eigenschaften, komparabel, sicher (de-)serialisierbar und ein Singleton zu sein, machen sie selbst dann zur ersten Wahl, wenn es gar nicht (mehr) um eigentliche Aufzählungslisten geht.

Literatur und Links

[Bloch21] J. Bloch, Effective Java Programming Language Guide, 1st Edition, Item 21: Replace enum constructs with classes, Addison-Wesley, 2001

[CFR] CFR, Another Java Decompiler, https://www.benf.org/other/cfr/

[JAPIEnum] Java Platform, Standard Edition 8 API Specification, Class Enum<E extends Enum<E>>, https://docs.oracle.com/javase/8/docs/api/java/lang/Enum.html

[JLSEnum] Java Language Specification, 8.9. Enum Types, https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.9

[JSCGSer] Secure Coding Guidelines for Java SE, 8 Serialization and Deserialization, https://www.oracle.com/technetwork/java/seccodeguide-139067.html#8

[JSON] JSON – JavaScript Object Notation, https://www.json.org

[Src] Quelltexte zum Herunterladen, https://link.simplexacode.ch/ser4