Ausgabe 02/2020

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 02

Java aktuell

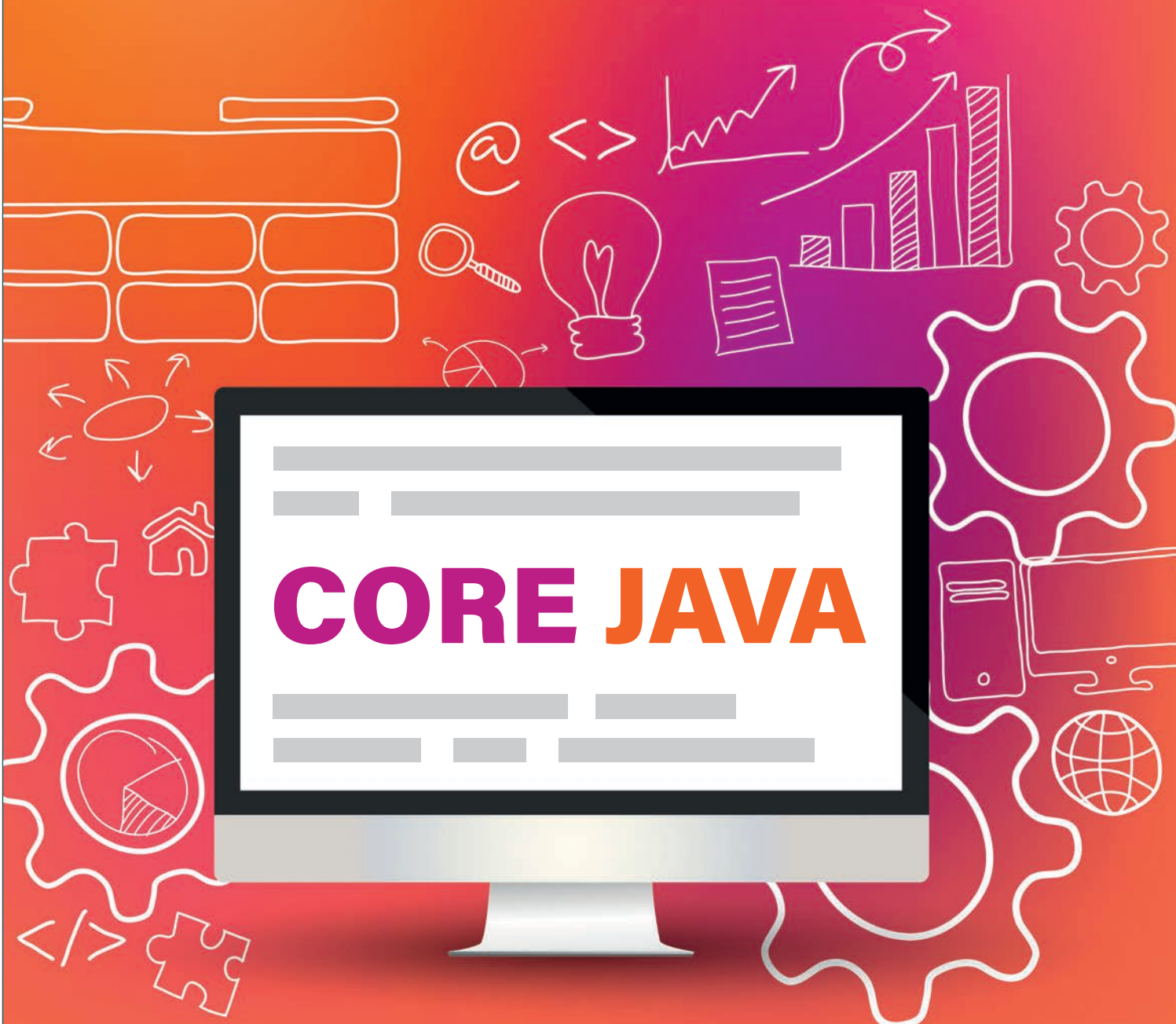
iJUG Verbund
www.ijug.eu

High Performance
Ein Blick hinter die Kulissen von Java

Immutability
Vorteile von unveränderlichen Datenstrukturen

Zeitreise
Die Highlights der Java-Versionen 8 bis 13 im Überblick

Das Herz des Java-Universums



DW *Made for minds.*

Coding for freedom.

Mehr Berufung als Beruf:
Dein IT-Job bei der DW.

JETZT BEWERBEN!
dw.com/it-karriere

Du entwickelst passgenau? Dann fordere uns auf der **JavaLand** am Klask heraus. Du willst was footern? Komm mit uns bei einer Tüte Popcorn ins Gespräch.

Du findest uns auf der **JavaLand** direkt gegenüber vom Haupteingang.

Immutable Java

Nicolai Mainiero, sidion

Viele Java-Entwickler schauen neidisch zu den Kollegen, die mit funktionalen Sprachen wie Scala oder Clojure Software schreiben. Diese müssen sich viel weniger Gedanken darüber machen, ob ihre Anwendungen auch dann noch korrekt sind, wenn sie parallel ausgeführt werden oder ob sie eine Datenstruktur bedenkenlos an eine Bibliothek übergeben können. Diese und noch weitere Vorteile ergeben sich daraus, dass die funktionalen Kollegen unveränderliche Datenstrukturen zur Verfügung haben. In diesem Artikel geht es darum, welche Vorteile sich daraus ergeben, wie unveränderliche Datenstrukturen in Java realisiert werden können und was es dabei zu beachten gibt.

Wenn man einen Java-Entwickler nach immutable Objekten in der Standardbibliothek fragt, fällt diesem wahrscheinlich sofort die `String`-Klasse ein. Dass es sich hierbei nicht um die einzige Klasse handelt, vergessen viele. Neben der `String`-Klasse sind auch alle „geboxten" primitiven Typen, wie zum Beispiel `Integer`, `Boolean` oder `Float`, immutable. Das moderne „Date and Time API" von Java, auch bekannt unter JSR-310 *[1]*, wurde explizit unter dem Gesichtspunkt der Immutability designt und implementiert. Darüber hinaus gibt es noch viele weitere in der Standardbibliothek verstreute Klassen, die immutable sind. Welche Vorteile ergeben sich nun daraus?

Vorteile von immutable Klassen

Immutable Objekte können Code vereinfachen und leichter verständlich machen. Sie führen oft auch zu einfacherem, sicherem und korrekterem Code, sind zuverlässige Schlüssel in HashMaps, sind Thread-safe und müssen nie kopiert oder geklont werden. In *Listing 1* ist das API eines Scheduler zu sehen, der eine Aufgabe (`task1`, `task2`) und einen Zeitpunkt, wann die entsprechende Aufgabe auszuführen ist, als Parameter akzeptiert. Je nach Implementierung kann der Scheduler die Aufgaben tatsächlich um einen Tag versetzt ausführen oder beide Aufgaben werden zum selben Zeitpunkt ausgeführt. Hier ist es also nötig, die Implementierung des Scheduler genau zu kennen, um zu beurteilen, ob der Code sich richtig verhält.

```
Date d = new Date();
Scheduler.scheduleTask(task1, d);
d.setTime(d.getTime() + ONE_DAY);
Scheduler.scheduleTask(task2, d);
```

Listing 1: Probleme beim Teilen von veränderbaren Objekten

Auch bei der Programmierung mit Threads kann man mit mutable Objekten Fehler machen, die nicht sofort offensichtlich sind. *Listing 2* demonstriert ein typisches Problem von nicht synchronisiertem Code.

In diesem Fall ist die `Sampler`-Klasse veränderbar und wird von beiden Threads gleichzeitig gelesen und beschrieben. Das führt dazu, dass die `incrementValue()`-Methode zwar insgesamt zwei Millionen Mal aufgerufen wird, der Wert von `value` aber in den meisten Fällen unter zwei Millionen bleiben wird. Ein etwas ungewöhnlicheres Problem, bei dem ein immutable Objekt Fehler verhindert, ist in *Listing 3* zu sehen. Wenn ein veränderbares Objekt als Schlüssel für eine HashMap verwendet wird, kann

```
public class Threads {

    public static void main(String[] args) throws InterruptedException {
        Sampler sampler = new Sampler();

        Thread thread1 = new Thread(new SamplerIncRunnable(sampler));
        thread1.start();

        Thread thread2 = new Thread(new SamplerIncRunnable(sampler));
        thread2.start();

        thread1.join();
        thread2.join();

        System.out.println(sampler.getValue());

    }

    static class SamplerIncRunnable implements Runnable {
        private Sampler sampler;

        public SamplerIncRunnable(Sampler sampler) {
            this.sampler = sampler;
        }

        public void run() {
            for ( int i=0; i<1_000_000; i++ ) {
                sampler.incrementValue();
            }
        }
    }

    static class Sampler {
        private int value = 0;

        void incrementValue(){
            this.value++;
        }

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

Listing 2: Probleme mit Nebenläufigkeit bei veränderbaren Objekten

es sein, dass nach einer Änderung des Felds des Schlüssels der zugehörige Wert nicht mehr gefunden wird. HashMap verwendet das Ergebnis der `hashCode()`-Methode, um den internen Schlüssel zu bestimmen.

Das Ändern des Namens der Person `jim` führt zu einer Änderung des `hashCode` dieses Objekts. Dadurch wird der Wert mit diesem Objekt als Schlüssel nicht mehr gefunden. Das Auflisten aller Einträge in der HashMap zeigt jedoch, dass das Objekt sich noch in der HashMap befindet.

Immutable Objekte erstellen

Die Vorteile von immutable Datenstrukturen sind nachvollziehbar und es bietet sich an, sie im täglichen Gebrauch zu nutzen. Java liefert alle notwendigen Mittel, um ein Objekt immutable zu definieren. In *Listing 4* sind zwei Klassen definiert, `Person` und `Address`. Eine Person hat einen Namen und eine Liste von Adressen. Die Adresse besteht in diesem vereinfachten Beispiel nur aus den Feldern „Stadt" und „Land".

Die Klassen enthalten Getter und Setter, wie sie üblich sind und von jeder IDE erzeugt werden können. Um eine Klasse immutable zu machen, sind folgende Schritte notwendig:

1. Deklariere die Klasse als `final`
2. Mache alle Felder `private` und `final`
3. Verhindere, dass der interne Zustand veränderbar nach außen gegeben wird
4. Mache im Konstruktor Kopien von veränderbaren Datenstrukturen

Listing 5 zeigt dieselben Klassen, nachdem diese vier Regeln angewendet worden sind, um die daraus entstehenden Objekte immutable zu machen. Den wesentlichen Teil der Arbeit kann uns hier die IDE abnehmen, wenn wir alle Felder der Klasse korrekt mit `final` deklariert haben. Dann werden zum Beispiel keine Setter mehr erzeugt und der erzeugte Konstruktor enthält alle Felder auch als Parameter.

Erwähnenswert in *Listing 5* ist der Konstruktor der Person, in dem mit `List.copyOf()` eine unveränderliche Kopie der übergebenen Liste erstellt wird. Diese Methode ist seit Java 10 verfügbar und ist der Methode `Collections.unmodifiableCollection()` vorzuziehen. Letztere liefert nur eine unveränderliche Ansicht auf die übergebene Collection zurück. Das bedeutet, dass wenn sich die darunterliegende Collection ändert, sich diese Änderung auf die erzeugte Ansicht auswirkt. In Java 9 kann man auf das Statement `List.of(collection.toArray())` ausweichen. Dies entspricht bis auf eine Speicheroptimierung der Implementierung von `List.copyOf()`.

```
import java.util.HashMap;
import com.google.common.truth.Truth;

public final class BadHashKey {

  final static class Person {
    private String name;

    public Person(String name) {
      this.name = name;
    }

    public String getName() {
      return name;
    }

    public void setName(String name) {
      this.name = name;
    }

    public int hashCode() {
      return name.hashCode();
    }

    public boolean equals(Object otherObj) {
      Person otherPerson = (Person) otherObj;
      return (name.equals(otherPerson.getName()));
    }
  }
    public static void main(String[] args) {
        Person jim  = new Person("James Holden");
        HashMap<Person, String> hashMap = new HashMap<>();
        hashMap.put(jim, "ice freighter");
        Truth.assertThat(hashMap.get(jim)).contains("ice freighter");
        jim.setName("Julie Mao");
        hashMap.entrySet().forEach(
          e -> System.out.println(e.getKey().getName() + ", " + e.getValue())
        );
Truth.assertThat(hashMap.get(jim)).contains("ice freighter"); // fails
    }
}
```

Listing 3: Probleme mit veränderbaren Objekten als Schlüssel

Veränderungen während der Softwareentwicklung

Immutable Objekte haben viele Vorteile und sind mit Java-Bordmitteln leicht zu realisieren. Allerdings erfordern sie ein gewisses Umdenken bei der Softwareentwicklung. Durch das bewusste Entfernen der Setter können bestehende Objekte nicht mehr manipuliert werden. Um dennoch Änderungen an Objekten, also neue Objekte mit geänderten Werten, vorzunehmen, wird eine andere Methode erforderlich. Üblich ist, die sogenannten Wither-Methoden als Pendants zu Settern zu schreiben, um eine Kopie eines Objekts zu bekommen, in dem ein Feld geändert worden ist. In *Listing 6* wird dieses Pattern vorgestellt. Das Prinzip dieser Wither-Methoden ist simpel: Für jedes Feld der Klasse wird eine `withX()`-Methode erzeugt. In dieser wird ein neues Objekt vom selben Typ erstellt und alle Felder bis auf X werden aus dem aktuellen Objekt übernommen. Der neue Wert für X wird der Methode als Parameter übergeben.

```
import java.util.List;

public class Person {
    private String name;
    private List<Address> addresses;

    public Person() {
        super();
    }

    public Person(String name, List<Address> addresses) {
        this.name = name;
        this.addresses = addresses;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }


    public List<Address> getAddresses() {
        return addresses;
    }

    public void setAddresses(List<Address> addresses) {
        this.addresses = addresses;
    }
}

public class Address {

    private String city;
    private String country;

    public Address() {
        super();
    }

    public Address(String city, String country) {
        this.city = city;
        this.country = country;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }
}
```

Listing 4: Zwei einfache veränderbare Klassen

Das Erzeugen neuer Objekte wie hier ist nicht langsamer, als ein bestehendes Objekt zu verändern. Tatsächlich sind die aktuellen Garbage Collector darauf optimiert, mit einer großen Menge kurzlebiger Objekte umzugehen. Dieser Code-Stil hat daher kaum Auswirkung auf die Performance *[3]*. Eine weitere Anpassung ist die konsequente Verwendung des Builder-Patterns bei Objekten mit mehr als drei Feldern. Durch die Verwendung von benannten Methoden zur Initialisierung der Objekte werden Fehler, zum Beispiel durch eine falsche Reihenfolge der Parameter, vermieden. Wenn gewünscht, kann dieser Stil auch mithilfe des Projekts Lombok *[4]* erreicht werden. Es müssen dann folgende Annotationen verwendet werden, um dasselbe Ergebnis zu erreichen. Erstens `@Value`, um die Klasse immutable zu machen. Lombok deklariert dann bei der Erzeugung der Implementierung alle Felder `private` und `final`, egal wie sie im Quellcode definiert sind. Die Klasse wird ebenfalls `final` markiert. Zweitens `@Builder`, um einen Builder für die Klasse zu erzeugen. Dieser macht den Konstruktor von `@Value` dann ebenfalls `private`, da dieser dann nur noch vom Builder verwendet werden soll. Und zuletzt `@With`, um die Wither-Methoden zu erzeugen, die es erlauben, Kopien des Objekts mit einzelnen geänderten Feldern zu erzeugen.

Immutable Objekte mit Immutables

Allerdings gibt es auch Vorbehalte gegenüber Lombok, da es den Quellcode während des Kompilierens erzeugt beziehungsweise verändert. Da diese Änderungen durch nicht öffentliche APIs gemacht werden, kann es sein, dass es bei einem neuen JDK Release nicht mehr funktioniert. Ebenfalls muss für jede IDE eine Extra-Erweiterung geschrieben werden, damit Funktionen wie Auto-Complete nutzbar sind.

Immutables *[5]* versucht, genau diese Probleme zu vermeiden und einfache, sichere und konsistente immutable Objekte zu erzeugen. Dazu integriert es sich als Annotation-Prozessor und erzeugt aus annotierten Interfaces konkrete Implementierungen, die dann eben auch inspiziert werden können. Es ist dasselbe Prinzip, das auch MapStruct *[6]* verwendet. In *Listing 7* ist zu sehen, wie kompakt sich ein Mitarbeiter mit Namen und Alter mit Immutables definieren lässt.

Der Annotation-Prozessor erzeugt aus dieser Definition neben einem `Builder` für jedes Attribut eine Wither-Methode und natürlich sinnvolle `hashCode()`-, `equals()`- und `toString()`-Methoden. Die Verwendung der so erzeugten Klassen ist einfach und unkompliziert, wie in *Listing 8* zu sehen ist.

Einen Nachteil darf man dennoch nicht verschweigen. Beim Zusammenspiel mit APIs von anderen Frameworks muss man oft wieder auf veränderbare Objekte ausweichen. So müssen zum Beispiel alle Entitäten bei JPA einen Default-Konstruktor besitzen. Dies verhindert, dass alle Felder als `final` deklariert werden können, und macht die Klasse mutable. Hier bietet es sich an, eine klare Grenze zwischen den unveränderlichen und den veränderlichen Objekten zu definieren.

```
public class Address {

    private final String city;
    private final String country;

    public Address(String city, String country) {
        this.city = city;
        this.country = country;
    }

    public String getCity() {
        return city;
    }

    public String getCountry() {
        return country;
    }
}
public class Person {
    private final String name;
    private final List<Address> addresses;

    public Person(final String name, final List<Address> addresses) {
        this.name = name;
        this.addresses = List.copyOf(addresses); // <1>
    }

    public String getName() {
        return name;
    }

    public List<Address> getAddresses() {
        return addresses;
    }
}
```

Listing 5: Immutable Person und Address

```
public class Person {
    private final String name;
    private final String address;


    public Person(final String name, final String address) {
        this.name = name;
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public Person withName(final String newName){
        return new Person(newName, address);
    }
}
```

Listing 6: Immutable Objekt mit Wither-Methoden

```
@Value.Immutable
abstract static class Employee {
public abstract String name();
public abstract Integer age();
}
```

Listing 7: Immutable Employee mit Immutables

Fazit

In „Effective Java“ *[7]* schreibt Josuha Bloch: „Classes should be immutable unless there's a very good reason to make them mutable.... If a class cannot be made immutable, limit its mutability as much as possible.“ Genau das sollte auch der Anspruch sein, wie in Java mit Immutability umgegangen wird. So kann man einen Anwendungskern entwickeln, der die Fachlogik enthält und immutable ist. Um diesen herum werden die Bibliotheken von Dritten, wie zum Beispiel JPA oder eine REST-Schnittstelle, angeordnet. Dieses Vorgehen führt zu einer sauberen Architektur, die von den Vorteilen unveränderlicher Klassen profitiert, ohne dass Workarounds gebaut werden müssen, um nicht immutable Bibliotheken an den eigenen Code anzupassen.

Quellen

[1] JSR 310: Date and Time API (2014), *https://jcp.org/en/jsr/detail?id=310*

[2] The Java Tutorials > Immutable Objects, *https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html*

[3] Brian Goetz (2003): Urban performance legends, *https://www.ibm.com/developerworks/library/j-jtp04223/index.html*

[4] Projekt Lombok, *https://projectlombok.org/*

[5] Immutables, *https://immutables.github.io/*

[6] MapStuct, *https://mapstruct.org/*

[7] Joshua Bloch (2017): Effective Java. Addison-Wesley Professional, Boston

[8] Quellcode unter *https://github.com/nicolaimainiero/immutable-java*

Nicolai Mainiero

sidion

nicolai.mainiero@sidion.de

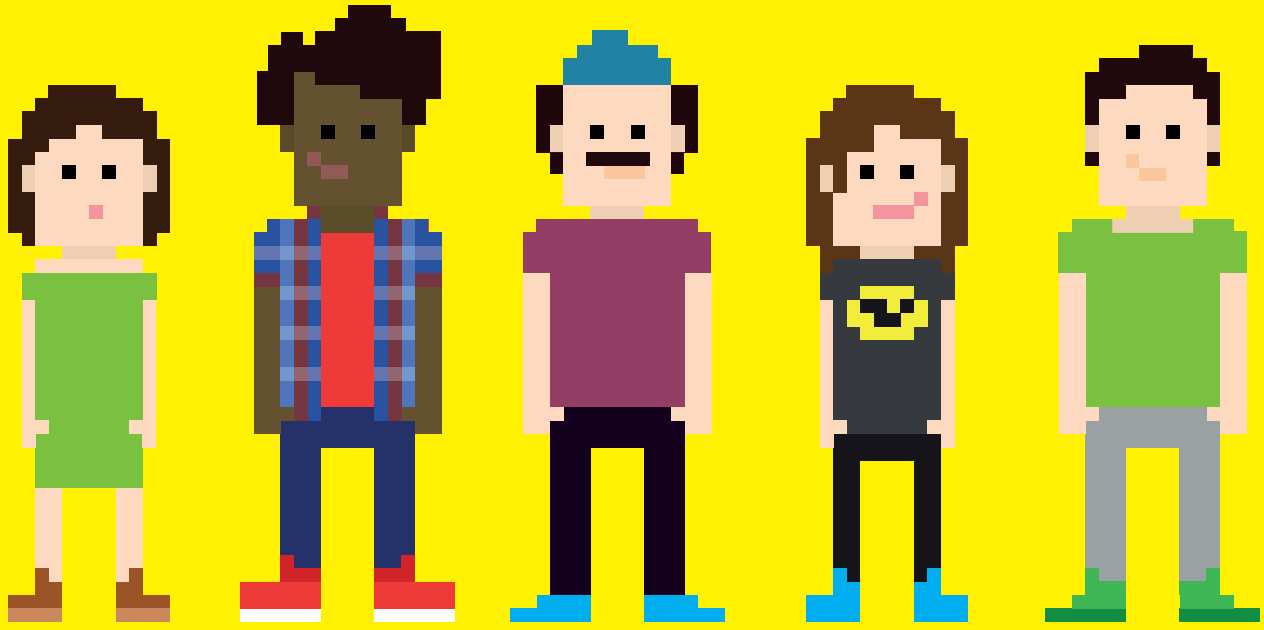
Nicolai Mainiero ist Diplom-Informatiker und arbeitet als Software Developer bei der Firma sidion. Er entwickelt seit über zwölf Jahren Geschäftsanwendungen in Java, Kotlin und PHP für unterschiedlichste Kundenprojekte. Dabei setzt er vor allem auf agile Methoden wie Kanban. Außerdem interessiert er sich für funktionale Programmierung, Microservices und reaktive Anwendungen.

```
final Employee julie = ImmutableEmployee.builder().name("Julie Mao").age(28).build();
final Employee james = ImmutableEmployee.copyOf(julie).withName("James Holden");
Truth.assertThat(julie.age()).isEqualTo(28);
Truth.assertThat(james.name()).isEqualTo("James Holden");
```

Listing 8: Immutable Employee mit Immutables

HAST DU ES SCHON MIT EINEM NEUSTART VERSUCHT?

Come and join the CI Crowd.

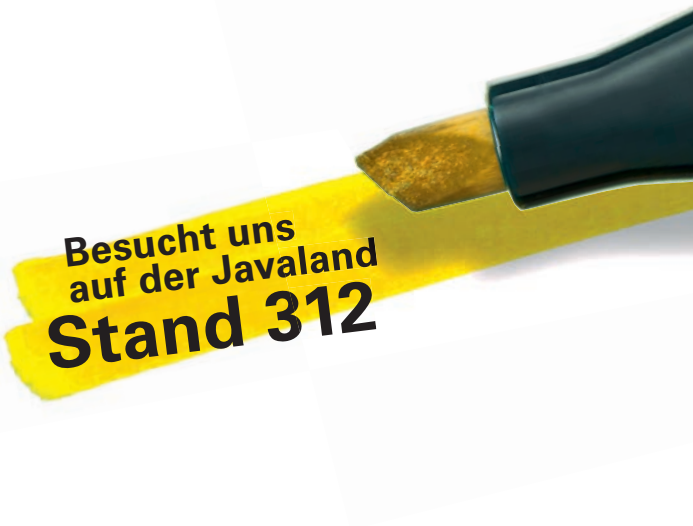


JAVA DEVELOPER BI BERATER UX DESIGNER MOBILE DEVELOPER DU

Hier neu starten:
karriereportal.
cologne-intelligence.de

COLOGNE INTELLIGENCE

QAWARE
SOFTWARE ENGINEERING



IT-Probleme lösen.
Digitale Zukunft gestalten.

Mit Erfindergeist
und Handwerksstolz.

Great Place To Work.
2019
Beste Arbeitgeber™
Information & Telekommunikation

kununu.com/qaware
qaware.de/karriere