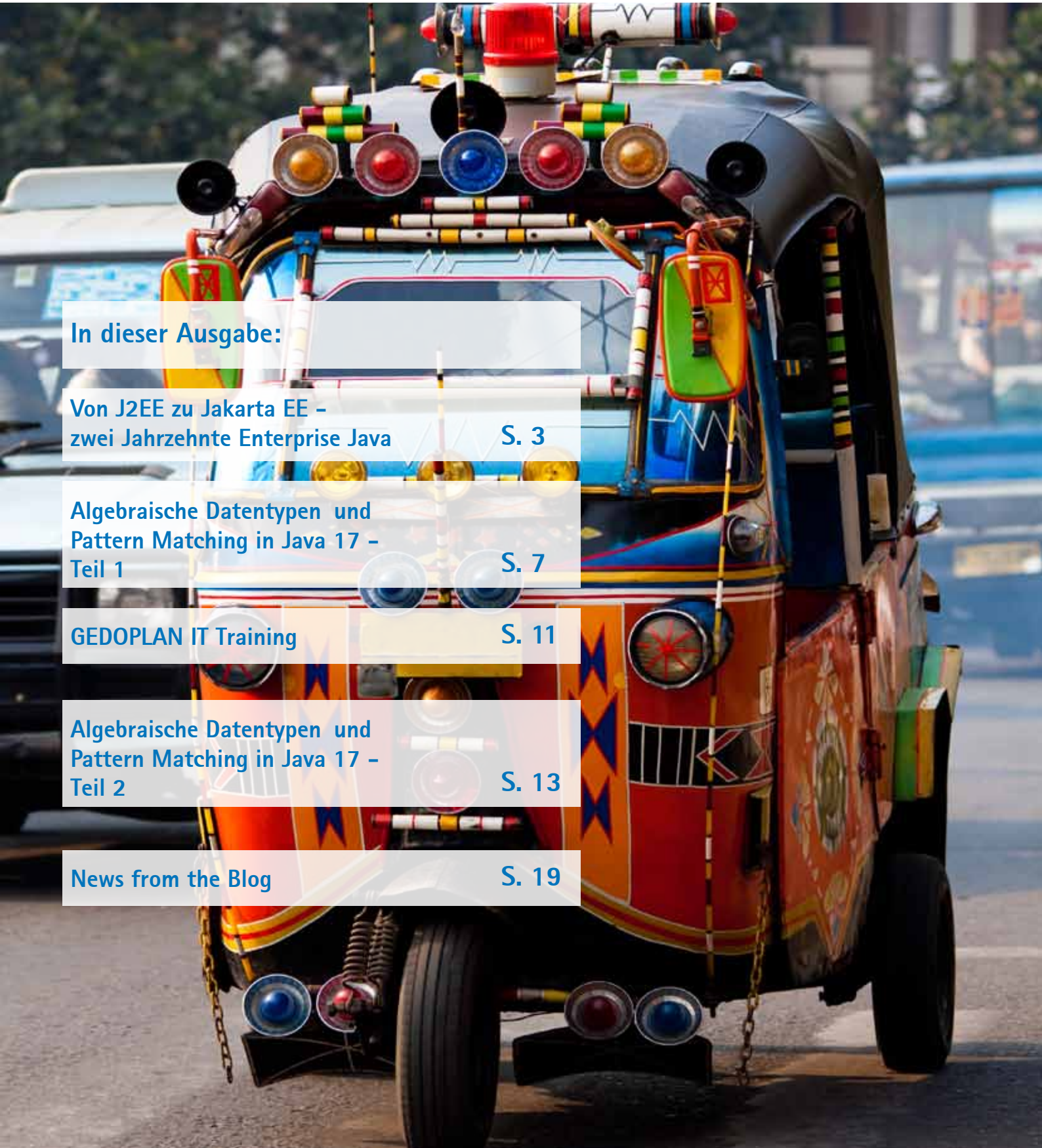
Ausgabe Frühjahr 2022

www.gedoplan.de

GEDOPLAN aktuell



In dieser Ausgabe:

In dieser Ausgabe finden sich ausschließlich Motive Jakartas.
Links das typische Transportmittel: ein Tuk Tuk, eine dreirädrige Autorikscha.

Liebe Leserin, lieber Leser,

vor etwas mehr als 22 Jahren veröffentlichte Sun die erste Version der Java Enterprise Edition, mittlerweile Jakarta Enterprise Edition. Unser Geschäftsführer Dirk Weil schaut zurück und zeichnet die wichtigsten Meilensteine dieser Entwicklung nach.
Mittlerweile wird halbjährlich ein LTS-Release der Programmiersprache Java veröffentlicht. Da kann man leicht den Überblick über die Neuigkeiten verlieren. Jens Seekamp beschränkt sich dabei in seinem zweiteiligen Artikel über Java 17 auf die Unterstützung von algebraischen Datentypen und das darauf basierende Pattern Matching. Er glaubt, dass ähnlich wie bei den Lambda-Ausdrücken man es erst verstehen muss, dann sich daran durch tägliche Praxis gewöhnen, um es schließlich einfach cool zu finden.
Das Open Source Server Framework Quarkus von Red Hat stellt aktuell einen Schwerpunkt bei GEDOPLAN IT Training dar. Wir stellen Ihnen unser Schulungsangebot zu diesem Thema vor, bei dem wir mittlerweile auf umfassende Erfahrung zurückgreifen können. (*www.gedoplan.de/quarkus)*
Wir haben zudem einen Schulungskonfigurator für Sie entwickelt, mit dem Sie komfortabel Ihre Schulung individuell zusammenstellen können. *(www.gedoplan.de/firmenschulungen)*
Und wie immer finden Sie am Schluss eine kleine Zusammenfassung der Tipps und Tricks aus unserem Java Blog.

Viel Spaß beim Lesen!

Ulrich Hake | ulrich.hake@gedoplan.de

Ulrich Hake

Termine

Expertenkreis Java

Thema: **Jakarta EE 10: Ein erster Blick auf Eclipse JNoSQL**
Ort: remote
Termin: Donnerstag, 28.04.2022 | 18:00 - 19:30 Uhr
Referent: Markus Pauer

Thema: **Der Application Server ist tot (?) – es lebe Jakarta EE!**
Ort: remote
Termin: Donnerstag, 17.03.2022 | 18:00 - 19:30 Uhr
Referent: Dirk Weil

Thema: **Was geht mit Java 17?**
Ort: remote
Termin: Donnerstag, 20.01.2022 | 18:00 - 19:30 Uhr
Referent: Jens Seekamp

Vorträge

Thema: **Java on Tracks – Modellbahnsteuerung mit Dirk Weil**
Ort: Java User Group Oberpfalz
Termin: 07.12.2022
Referent: Dirk Weil, GEDOPLAN GmbH

Thema: **JEE-Anwendungen auf Quarkus migrieren**
Ort: Java User Group Hamburg
Termin: 31.05.2022
Referent: Dirk Weil, GEDOPLAN GmbH

Thema: **Der Application Server ist tot (?) – es lebe Jakarta EE!**
Ort: JavaLand
Termin: 16.03.2022
Referent: Dirk Weil, GEDOPLAN GmbH

Von J2EE zu Jakarta EE – zwei Jahrzehnte Enterprise Java

Enterprise Java wird in diesem Jahr 22 - zumindest wenn man den Standard betrachtet. Dabei gab es verschiedene Lizenzeigentümer, viele Expert Groups, teilweise revolutionäre Entwicklungen, aber auch vermeintlichen Stillstand und Ungewissheit. Werfen wir mal einen Blick auf die bewegte Vergangenheit.

Von Dirk Weil

J2EE

Sun veröffentlichte Ende 1999 die Java 2 Platform, Enterprise Edition - kurz J2EE. Bemerkenswert waren daran einige Dinge.
So war die J2EE eine Umbrella Specification, die diverse Einzelstandards umfasste, die zwar unabhängig entwickelt, aber dennoch koordiniert unter einem Dach zusammengeführt wurden. Zudem gab es sowohl für die Bestandteile als auch für das Ganze direkt mehrere Implementierungen - und nicht nur von Sun. Wenngleich die Lizenz und auch die Produkte im Allgemeinen nicht so waren, sehen wir hier schon Ansätze wie in vielen Open Source-Projekten.
Mit Enterprise JavaBeans war ein Standard enthalten, der die Welt stark polarisierte. Zum einen konnten hiermit kleine Geschäftslogikeinheiten deklarativ miteinander kombiniert und mit verschiedenen Scopes und Transaktionsmodi versehen werden - aus heutiger Sicht für 1999 geradezu revolutionär. Zum anderen platzte der Standard vor technischer Komplexität aus allen Nähten. Die Komponenten bestanden aus mehreren Klassen und Interfaces, die von bestimmten Basistypen abzuleiten waren und deren Methoden sehr technische Signaturen haben mussten. Für die o. a. Konfiguration und Verknüpfung waren recht komplexe XML-Deskriptoren beizusteuern.
Um bspw. zwei simple statuslose Services zu programmieren, die zur Demonstration nur Texte produzieren, wobei der eine den zweiten aufruft, sind sage und schreibe je 3 Java-Quellen nötig:

```
// WorldService.java
public interface WorldService extends EJBLocalObject {
  public String getWorld();
}

// WorldServiceHome.java
public interface WorldServiceHome extends EJBLocalHome {
  public WorldService create()
    throws RemoteException, CreateException;
}

// WorldServiceBean.java
public class WorldServiceBean implements SessionBean {

  public String getWorld() { return "world"; }

  public void ejbCreate()
    throws RemoteException, CreateException {}
  public void setSessionContext(SessionContext ctx)
    throws EJBException, RemoteException {}
  public void ejbRemove()
    throws EJBException, RemoteException {}
  public void ejbActivate()
    throws EJBException, RemoteException {}
  public void ejbPassivate()
    throws EJBException, RemoteException {}
}

// HelloService.java
public interface HelloService extends EJBLocalObject {
  public String getHello();
}

// HelloServiceHome.java
public interface HelloServiceHome extends EJBLocalHome {
  public HelloService create()
    throws RemoteException, CreateException;
}

// HelloServiceBean.java
public class HelloServiceBean implements SessionBean {

  private WorldService worldService;

  public String getHello() {
    return "Hello, " + this.worldService.getWorld();
  }

  public void ejbCreate()
    throws RemoteException, CreateException {
    try {
      InitialContext context = new InitialContext();
      WorldServiceHome worldServiceHome = (WorldServiceHome)
        context.lookup("java:comp/env/WorldService");
      this.worldService = worldServiceHome.create();
    } catch (Exception e) {
      throw new CreationException(e);
    }
  }

  public void ejbCreate()
    throws RemoteException, CreateException {}
  public void setSessionContext(SessionContext ctx)
    throws EJBException, RemoteException {}
  public void ejbRemove()
    throws EJBException, RemoteException {}
  public void ejbActivate()
    throws EJBException, RemoteException {}
  public void ejbPassivate()
    throws EJBException, RemoteException {}
}
```

Dazu kommt dann noch ein Deployment Descriptor:

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://
java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd" version="2.1">
  <enterprise-beans>
    <session>
      <ejb-name>WorldService</ejb-name>
        <local-home>de.gedoplan.showcase.service.WorldService-
Home</local-home>
      <local>de.gedoplan.showcase.service.WorldService</local>
        <ejb-class>de.gedoplan.showcase.service.WorldServiceBe-
an</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
    <session>
      <ejb-name>HelloService</ejb-name>
        <local-home>de.gedoplan.showcase.service.HelloService-
Home</local-home>
      <local>de.gedoplan.showcase.service.HelloService</local>
        <ejb-class>de.gedoplan.showcase.service.HelloServiceBe-
an</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <ejb-local-ref>
        <ejb-ref-name>WorldService</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <local-home>de.gedoplan.showcase.service.WorldService-
Home</local-home>
        <local>de.gedoplan.showcase.service.WorldService</local>
      </ejb-local-ref>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Der überwiegende Teil davon ist Boiler Plate Code - nur ca. 5 % sind "Fachlogik"!

Java EE

Das konnte natürlich so nicht weitergehen. Die Idee zur Vereinfachung kam vom zeitparallel entwickelten Framework Spring: Dependency Injection. Statt jede Komponente ihr eigenes Setup und die eventuell benötigten Verknüpfungen zu anderen Komponenten selbst erledigen zu lassen, übernimmt dies ein Container in der Server-Runtime. Die technischen Aspekte der Komponenten verschwinden damit aus dem Fokus der Entwicklung. Keine (verpflichtenden) Lifecycle-Methoden mehr, keine Home-Interfaces. Wenn Sie nicht wissen, was das war und wofür man das benötigte - vergessen Sie's einfach!
In Java EE 5 übernahm man diese Idee aus Spring und ging sogar noch einen Schritt weiter: Statt die Konfiguration weiterhin in XML vorzunehmen, konnten nun Annotationen genutzt werden. Die oben angerissenen Beispiel-Services sehen nun so aus:

```
// WorldService.java
@Stateless
public class WorldService {
  public String getWorld() {
    return "world";
  }
}
// HelloService.java
@Stateless
public class HelloService {
  @EJB
  WorldService worldService;

  public String getHello() {
    return "Hello, " + this.worldService.getWorld();
  }
}
```

Java EE 5 wurde in 2006 veröffentlicht. Dass dabei der Name von J2EE in Java EE geändert wurde, war eine Folge der entsprechenden Umbenennung der Sprache Java (und die 2 im Namen hatte vermutlich niemand wirklich verstanden). Zudem wurde die Nummerierung von 1.x in x geändert (schließlich war .NET schon bei 6).

Java EE reloaded

Eigentlich hätte es damit nun gut sein können, aber die oft gehörte Meinung über EJB (und damit ganz häufig über Java EE insgesamt) war "Das ist alles zu schwergewichtig. In Spring geht das mit viel weniger Code".
Häh? Ersetzen wir im Code oben @Stateless durch @Service und @EJB durch @Autowired, haben wir Spring-Code. Gleiche Länge, gleiche Komplexität.
Es ist vermutlich wie mit anderen Dingen des menschlichen Lebens: Ist der Ruf einmal versaut, bleibt der Makel auf ewige Zeit haften. EJB ist durch die frühen Versionen verbrannt - keine Rettung mehr möglich!
Was macht man dann? Richtig: Man gibt dem Kind einen anderen Namen. So wurde in Java EE 6 der neue Standard CDI eingebaut. Damit könnten die beiden Services nun diesen Code haben:

```
// WorldService.java
@ApplicationScoped
public class WorldService {
  public String getWorld() {
    return "world";
  }
}

// HelloService.java
@ApplicationScoped
public class HelloService {
  @Inject
  WorldService worldService;
  public String getHello() {
    return "Hello, " + this.worldService.getWorld();
  }
}
```

Das fanden die Entwickelnden nun endlich leichtgewichtig. Interessant, wie viel "schwerer" @Stateless und @EJB im Gegensatz zu @ApplicationScoped und @Inject (oder @Service und @Autowired) zu sein scheinen ...
Ich würde CDI allerdings Unrecht tun, wenn ich es nur auf diese Annotationsänderungen reduzieren würde. In den vier Jahren zwischen Java EE 5 und 6 sind vielmehr viele weitere Ideen in CDI eingeflossen, die EJB vorher nicht besaß.

Problematisch ist an der Koexistenz von EJB und CDI im Gesamtstandard, dass nun zwei Bausteine für denselben Zweck existieren. Es sollte jedoch für jede Aufgabe stets nur ein Angebot vorhanden sein. Um dies abzumildern, muss EJB alle CDI-Eigenschaften kompatibel implementieren. Komponenten beider Frameworks können somit in einer Anwendung genutzt und problemlos miteinander kombiniert werden. In der Zukunft wird EJB aber eine zunehmend kleinere Rolle spielen.

Jakarta EE

Die Plattform wurde in der Folge für die Releases 7 und 8 weiterentwickelt, aber in den Jahren 2016 und 2017 entstand eine irritierende Pause, als insbesondere die von Oracle geleiteten Teilspezifikationen nicht voranzukommen schienen. Der sehr hierarchische Standardisierungsprozess JCP (Java Community Process) verlangt die Mitwirkung des (einen) Specification Leads, so dass dieser durch Inaktivität die Weiterentwicklung faktisch stoppen kann.
Die Java-Community war sich nicht sicher, was da passiert - oder eben auch nicht passiert. Würde Java EE eventuell sogar aufgegeben, "eingestampft"? Die Erlösung folgte 2017, als einerseits Java EE 8 veröffentlicht wurde und andererseits Oracle verkündete, dass die

Straßenmarkt in Jakarta

Plattform als Open Source-Projekt an die Eclipse Foundation gehen würde.
Dieser guten Nachricht folgte leider zweimal "Aber": Oracle behielt zum einen die Rechte am Namen Java EE. Im Open Source-Projekt entschied man sich mit Hilfe eines Community Votings für den neuen Namen Jakarta EE. Du liebe Güte: Der dritte Name innerhalb überschaubarer Zeit. Das verwirrt schon mal den einen oder anderen, der Fachkräfte für Projekte sucht. Immerhin kann man immer noch die Abkürzung JEE verwenden.
Dramatischer noch ist die Tatsache, dass sich Oracle und Eclipse Foundation nicht auf eine weitere Nutzung des im Enterprise Java häufig auftretenden Paketnamen-Präfixes javax einigen konnten. Die bisherigen Klassen und Interfaces dürfen zwar in Jakarta EE 8 verwendet werden, aber jede Weiterentwicklung in diesem Namensraum ist untersagt. Die Folge: In einem Big Bang werden sämtliche JEE-Pakete von javax.XYZ in jakarta.XYZ umbenannt. Was nach einem einfachen Textersatz aussieht bedeutet in der Praxis, dass ältere Anwendungen und Bibliotheken auf neueren Servern oder in neueren Runtime-Frameworks nicht mehr ohne Weiteres lauffähig sein werden. Es deutet sich an, dass Server das Problem durch eine Modifikation des Bytecodes angehen werden, also praktisch beim Laden von Anwendungen die betroffenen Paketnamen on-the-fly anpassen werden.
In den Umbenennungs-Reigen gehören noch die XML-Namespaces (aus http://xmlns.jcp.org wird https://jakarta.ee) und Konfigurations-Properties (z. B. javax.persistence.jdbc.driver --> jakarta.persistence.jdbc.driver).
Da das Paket-Präfix nicht nur im JEE-Bereich genutzt wird, kann man nicht etwa in seinen Programmquellen javax. durch jakarta. ersetzen. Glücklicherweise unterstützen die gängigen IDEs recht gut bei der Migration. Ob Oracle bewusst war, welchen immensen Aufwand das Festhalten an Java EE und javax weltweit erzeugen wird?
Durch die beschriebenen Legal Issues ergibt sich die folgende Release-Situation:

- Jakarta EE 8 wurde 2019 code-gleich zu Java EE 8 veröffentlicht. Geändert haben sich ausschließlich die Spezifikationsdokumente, in denen die dem Urheberrecht von Oracle unterliegenden Namen durch neue ersetzt wurden.
- Jakarta EE 9 wurde 2020 mit unveränderter Funktionalität released, wobei die zuvor beschriebenen Umbenennungen von Paketen, Namespaces und Properties durchgeführt wurden.
- Jakarta EE 10 sollte ursprünglich Ende 2021 oder Anfang 2022 veröffentlicht werden. Nun wird es vermutlich das Q2 2022 werden. Es ist das erste Release seit Java EE 8, das Änderungen der Funktionalität einführt. Die Details sind noch nicht ganz finalisiert, aber relativ klar ist, dass CDI um nahezu alle Features ergänzt wird, die bislang nur in EJB zu finden sind. Zudem wird eventuell Config aus MicroProfile nach Jakarta EE übernommen. Gute Chancen hat auch die Integration von NoSQL..

Der Übergang zu Jakarta EE hat organisatorisch einige Vorteile erzeugt. Der Standardisierungsprozess ist öffentlich und community-driven mit den üblichen Werkzeugen wie Git, Pull Requests, AsciiDoc etc. Wer mag und sich berufen fühlt, ist eingeladen mitzumachen - ohne große Hürden. Die TCKs (Technology Compatibility Kits aka Tests), die früher Closed Source waren, sind nun frei verfügbar, sodass viel leichter als zuvor funktionierende Implementierungen der Standards entstehen können. Es gibt auch nicht mehr die eine Reference Implementation, sondern Compatible Products.
In der nächsten Ausgabe werde ich einen Blick auf eben diese Produkte werfen, auf klassische (und dennoch leichtgewichtige (!)) Application Server, aber auch auf JAR Deployments und Micro(profile) Runtimes.

Weitere Informationen

Wir versorgen Sie gerne mit weiterem Input:

- Unser Expertenkreis Java ist eine regelmäßige Vortragsveranstaltung zu allem aus dem Java-Ökosystem. Melden Sie sich kostenfrei an unter *https://gedoplan.de/java-events/*.
- In die Tiefe gehen wir in unseren Seminaren, z. B. mit dem **Power Workshop Jakarta EE**, zu finden auf *https://gedoplan.de*.
- In loser Folge finden Sie auch in unserem Blog Neuerungen, kleine und große Dinge rund um Java, Einstieg wiederum über *https://gedoplan.de*.

...

Dirk Weil [dirk.weil@gedoplan.de]
- Geschäftsführer der GEDOPLAN GmbH -
Fachbuchautor, schreibt Artikel für Fachmagazine, hält Vorträge und leitet Seminare und Workshops zu diversen Java-SE-/EE-Themen

Algebraische Datentypen und Pattern Matching in Java 17 – Teil 1

Mit Java 17 steht uns seit September 2021 ein aktuelles LTS-Release der Programmiersprache Java zur Verfügung. Und nun steht schon mit Java 18 im März 2022 ein neues Release vor der Tür. Durch den halbjährlichen Release-Zyklus ergibt sich eine Vielzahl von JEPs (Java Enhancement Proposal) für die jeweiligen Sprach-Versionen. Durch die vielen JEPs inkl. Incubator- bzw. Preview-Features verliert man vielleicht den Überblick bezüglich der bedeutenderen Sprach-Entwicklungen. Deshalb wollen wir hier einmal den Blick auf eine solche richten: Die Unterstützung von algebraischen Datentypen und das darauf basierende Pattern Matching.

Von Jens Seekamp

Überblick: JEPs und Sprach-Konstrukte

Die Einführung von algebraischen Datentypen und Pattern Matching hat mit Java 14 begonnen und ist mit Java 17 noch nicht abgeschlossen. Algebraische Datentypen wurden in Java als **Records** und **Sealed Classes** eingeführt. Die vom **Pattern Matching** betroffenen Sprach-Konstrukte sind derzeit der instanceof-Operator sowie die switch-Anweisung bzw. -Ausdruck. Insofern können auch die Neuerungen zur mehrfachen Fallunterscheidung als Vorläufer der hier behandelten Sprach-Entwicklungen mit einbezogen werden. Umgekehrt wird an der Erweiterung des Pattern Matching in zukünftigen Java-Versionen gearbeitet.
Somit ergibt sich folgende Liste von hier relevanten JEPs, wobei die verschiedenen Preview-Vorstufen nicht mit aufgeführt sind (s. [1]):

- JEP 361: Switch Expressions (Java 14)
- JEP 395: Records (Java 16)
- JEP 409: Sealed Classes (Java 17)
- JEP 394: Pattern Matching for instanceof (Java 16)
- JEP 406: Pattern Matching for switch (Preview in Java 17)
- JEP 405: Record Patterns (Preview, bislang ohne Release-Zuordnung)

Teil 1: Algebraische Datentypen

Was sind algebraische Datentypen?

In der Mathematik wird unter einer **algebraischen Struktur** eine Trägermenge zusammen mit Operationen auf den Elementen der Trägermenge verstanden. Ein Beispiel - welches wir alle aus Schule oder Studium kennen - sind die reellen Zahlen REAL als Trägermenge mit der Addition + und der Multiplikation x als Operationen. Auf dieser algebraischen Struktur können wir Gleichungen wie z. B. a = 2 x b + 3,25 definieren, welche aus Zahlen, Variablen und Operatoren bestehen.
Dieses Konzept einer algebraischen Struktur überträgt man nun auf das **Typsystem einer Programmiersprache**. Die Trägermenge TYPES ist die Menge der Datentypen der Programmiersprache. Jede Programmiersprache hat bestimmte vordefinierte Datentypen: in Java beispielsweise Boolean, Integer, Double oder String. Als Operationen auf den Datentypen der Trägermenge TYPES werden der Summentyp +und der Produkttyp x eingeführt. Damit können wir quasi Datentyp-Gleichungen notieren, die sich analog zu oben Gesagtem aus Literalen (d. h. bereits definierten Datentypen), Typ-Variablen (z. B. T, vgl. Generics in Java) und den Typ-Operatoren + bzw. x zusammensetzen. Das Konzept der algebraischen Datentypen wird vor allem in funktionalen Programmiersprachen wie beispielsweise Standard ML oder Haskell verwendet. Allgemein beschreiben algebraische Datentypen den Konstruktionsvorgang:

- wie auf Typ-Ebene aus vordefinierten Datentypen strukturierte Datentypen definiert werden bzw.
- wie auf Wert-Ebene aus einfachen Werten zusammen gesetzte Werte gebildet werden.

Produkttypen

Ein **Produkttyp** P fasst mehrere Datenobjekte von Datentypen Ti (i=1,…,n) zu einem Wert zusammen. Dies entspricht mathematisch dem kartesischen Produkt über den Wertemengen der zugrundeliegenden Datentypen. Die entsprechende Datentyp-Gleichung lautet:

P = T1 x … x Tn

Hierbei werden die zum Produkttyp aggregierten Datentypen als **Komponenten** bezeichnet. Dies entspricht in Java ungefähr einer Klasse P, die aus ihren Attributen (d. h. Datenfeldern) vom Typ Ti ein neues Objekt vom Typ P konstruiert.

Beispiele für Produkttypen:

1. Ein Koordinatenpunkt als ein Paar (x,y) wird durch den Produkttyp Point = Double x Double definiert; dies entspricht der Java-Klasse java.awt.Point.
2. Ein Datum als ein 3-Tupel (day,month,year) wird durch den Produkttyp Date = Integer x Month x Integer definiert; dies entspricht der Java-Klasse java.time.LocalDate unter Verwendung von java.time.Month.
3. Ein selbstdefinierter Datentyp für eine Person als 4-Tupel (forename,surname,birthdate,gender) wird durch den Produkttyp Person = String x String x LocalDate x Gender mit einem selbstdefinierten Aufzählungstyp Gender definiert.

Wir erkennen, dass wir in Java - und auch jeder anderen jemals genutzten Programmiersprache - schon seit Jahr und Tag mit Produkttypen gearbeitet haben.
Einen Spezialfall stellt der **homogene Produkttyp** dar, welcher mehrere Datenobjekte eines Datentyps T zu einem Wert zusammenfasst. Dies entspricht mathematisch dem n-fachen kartesischen Produkt über der Wertemenge des zugrunde liegenden Datentyps. Die entsprechende Datentyp-Gleichung lautet:

C = T x ... x T (n-mal)

Beispiele für homogene Produkttypen:

1. Ein Array von Zeichenketten [s1,...,sn] wird durch den homogenen Produkttyp A = String x ... x String (n-mal) definiert; dies entspricht dem Java-Typ String[].
2. Eine Liste von ganzen Zahlen (z1,...,zn) wird durch den homogenen Produkttyp L = Integer x ... x Integer (n-mal) definiert; dies entspricht dem generischen Java-Typ java.util.List<E> mit dem aktuellen Typ-Parameter Integer.

Wiederum erkennen wir, dass wir schon seit langem in Java mit homogenen Produkttypen arbeiten; und zwar immer, wenn wir Datenobjekte in Arrays, Collections, Maps usw. sammeln. Eine Java-Map ist dabei ein homogener Produkttyp mit einem eingeschachtelten Produkttyp MapEntry = Key x Value für die Map-Einträge.

Records

Bisher haben wir herausgearbeitet, dass quasi jede Java-Klasse, die einen Zustand in Form von Attributen (d. h. Datenfeldern) einkapselt sowie Objekt-Sammlungen (d. h. [], Set, Properties usw.) Produkttypen sind. Die neu eingeführten Records sind daher lediglich ein Sprach-Konstrukt zur komfortablen Implementierung eines Produkttyps mit bestimmten Eigenschaften.
Ein **Record** ist eine Klasse, die im Sinne eines Produkttyps einen Objekt-Zustand mittels ihrer Attribute einkapselt. Jeder Record definiert eine (implizit) finale Klasse, deren Instanzen unveränderbar (immutable) sind. Ein Record wird mit dem neuen Schlüsselwort record anstelle des Schlüsselwortes class implementiert. Zwischen Record-Deklaration und -Rumpf werden in runden Klammern und durch Komma getrennt die Attribute des Produkttyps aufgelistet; dies erfolgt als eine Liste von Variablen-Deklarationen, wie wir es beispielsweise von Methoden-Parameterlisten gewohnt sind.

```
public record Person
        (String forename,
         String surname,
         LocalDate birthdate,
         Gender gender) {}
```

Records verfügen (implizit) über einen parametrisierten Konstruktor, Accessor-Methoden für den lesenden Zugriff auf die eingekapselten Attribute sowie weitere Standard-Funktionalität (z. B. Methode equals()). Dies erspart uns die Implementierung des lästigen Boiler-Plate-Code bzw. dessen Generierung mittels Java-IDE oder Lombok-Framework.

```
Person person = new Person(
                "Silke",
                "Musterfrau",
                LocalDate.of(1984,4,22),
                Gender.FEMALE);

String forename = person.forename();
```

Ansonsten verhalten sich Records wie reguläre Java-Klassen, d. h. sie können beispielsweise Interfaces implementieren, Typ-Parameter besitzen oder in ihrem Rumpf können weitere Attribute oder Methoden implementiert werden.
Darüber hinaus kann der letzte Attribut-Typ in einer Record-Deklaration ein variabler Typ sein, so dass z. B. die folgende Typ-Implementierung möglich wird:

```
public record NamedTuple(String name, Double... values) {}

new NamedTuple("Vektor_1", 0.5);
new NamedTuple("Vektor_2", 1.5, 4.234, 8.63);
```

Records unterstützen somit als nunmehr natives Sprachkonstrukt von Java die einfache Implementierung von unveränderbaren Daten-Behältern. Dies korrespondiert gut zum Building-Block *Value Object* im Domain-Driven Design (DDD). Auch das Entwurfsmuster *Data Transfer Object (DTO)* lässt sich mit Records denkbar effizient implementieren. Geradezu kontra-produktiv erscheint jedoch die Abkehr von der jahrzehntelangen Tradition der get-/set-Methoden. Natürlich hat ein Record als unveränderbares Objekt überhaupt keine set-Methoden. Jedoch bedingt der Übergang von der Getter-Namenskonvention (z. B. getForename()) zur Accessor-Namenskonvention (z. B. forename()) eine Unverträglichkeit mit den allermeisten Binding-Frameworks (z. B. JSON-B) und Persistenz-Lösungen (z. B. Jakarta Persistence mit Property-Access). Die Vorteile werden somit auf die lange Bank verschoben - wir müssen warten, bis die von uns im Projekt verwendeten Frameworks mit Records integriert sind.

Summentypen

Ein **Summentyp** S vereinigt unterschiedliche Datenobjekte von Datentypen Ti (i=1,...,n) zu einem generalisierten Wert. Dies entspricht mathematisch der disjunkten Vereinigung über den Wertemengen der zugrunde liegenden Datentypen. Die entsprechende Datentyp-Gleichung lautet:

S = T1 + ... + Tn

Hierbei werden die zum Summentyp vereinigten Datentypen als **Varianten** bezeichnet. Dies entspricht in Java ungefähr einer Oberklasse S, welche die gemeinsamen Eigenschaften von Typen Ti vereint.
Beispiel für Summentyp:
Eine geometrische Figur soll sein:

1. entweder ein Kreis, gegeben durch Mittelpunkt und Durchmesser
2. oder ein Rechteck, gegeben durch zwei Eckpunkte
3. oder ein Polygon, gegeben durch die Liste seiner Eckpunkte

Diese Typ-Modellierung wird durch die folgende Kombination von Summen- und Produkttypen realisiert:

Shape = Circle + Rectangle + Polygon

wobei

Circle = Point x Double
Rectangle = Point x Point
Polygon = Point x … x Point (homogen)

Wir erkennen, dass wir in Java durch die Bildung von Typ-Hierarchien (d. h. Nutzung von Vererbung) bereits mit Summentypen arbeiten. Ein wichtiger Unterschied liegt jedoch darin, dass ein Summentyp nur die definierten Varianten zulässt, wohingegen eine Oberklasse in Java eine nicht eingeschränkte Menge von Unterklassen haben darf. Der Summentyp Shape lässt somit eine Variante Square (Quadrat) nicht zu, jedoch kann in Java der Entwickler problemlos eine weitere Unterklasse Square zu der Oberklasse Shape implementieren.
Aufzählungstypen (enumeration) werden als Spezialfälle eines Summentyps interpretiert, bei dem jede unterschiedliche Variante genau einen Wert beinhaltet und dieser Wert durch den Datentyp selbst gegeben ist.

Beispiele für Aufzählungstypen als Summentyp (mit der in Java üblichen Großschreibung für Konstanten):

1. Die Menge der Wahrheitswerte {true,false} wird durch den Summentyp Boolean = TRUE + FALSE definiert; dies entspricht der Java-Klasse java.lang.Boolean.
2. Die Menge der Wochentage {monday,…,sunday} wird durch den Summentyp DayOfWeek = MONDAY + … + SUNDAY definiert; dies entspricht der Java-Enumeration java.time.DayOfWeek.
3. Für den Typ Person definieren wir in unserem Projekt analog Gender = MALE + FEMALE + …

Da sich schließlich auch noch **rekursive Datentypen** (z. B. Graphen, Binärbäume usw.) als spezielle Summentypen definieren lassen, erkennen wir, dass sich alle Datentypen innerhalb einer Programmiersprache mittels Produkt- und Summentypen darstellen lassen. In einem spartanischen Ansatz würde dies bedeuten, dass wir auch in Java bereits mit einer Menge TYPES = {Boolean,Integer,…} von vordefinierten Datentypen sowie dem Produkttyp x und dem Summentyp + für die Konstruktion zusammengesetzter Typen auskommen könnten.

Skyline Jakarta

Sealed Classes

Wir haben nun außerdem herausgearbeitet, dass in Java Ober-/Unterklassen sowie Aufzählungstypen (sowie rekursive Datentypen) quasi Summentypen sind. Die neu eingeführten Sealed Classes sorgen daher lediglich noch dafür, dass eine Vererbungs-Hierarchie auch tatsächlich einen Summentyp definieren kann. Der gewählte JEP-Titel *Sealed Classes* ist etwas irreführend, weil es neben versiegelten (sealed) Klassen auch versiegelte Interfaces geben kann. Zudem spielen auch Records und Aufzählungstypen eine gewisse Rolle. Daher sprechen wir hier verallgemeinert von einem versiegelten Typ.

Neben dem technischen Aspekt der Wiederverwendbarkeit kann und soll mit einer Vererbungs-Hierarchie auch eine fachliche Modellierung stattfinden. Insbesondere kann es aus fachlicher Sicht notwendig sein, nur bestimmte abgeleitete Typen zu erlauben, aber alle weiterhin denkbaren Untertypen auszuschließen. Dies wird mit den bisherigen Java-Sprachkonstrukten wie dem Schlüsselwort final (d. h. Ableitungsverbot) und dem Zugriffsmodifizierer package private (d. h. Untertypen aber auch Sichtbarkeit nur im selben Paket) nur unzureichend ermöglicht.
Bei einem **versiegelten Typ** handelt es sich um einen Java-Typ, der nur eine wohldefinierte Menge von abgeleiteten Typen zulässt. Dies erfolgt, indem die zulässigen, abgeleiteten Typen bei der Deklaration des versiegelten Typs explizit aufgelistet werden. Diese Steuerung der Vererbungs-Hierarchie hat zum einen eine möglichst allgemeine Nutzbarkeit und zum anderen eine Einschränkung der Ableitbarkeit von Typen zum Ziel.
Ein **versiegelter Typ** wird mit dem Typ-Modifizierer sealed deklariert. In diesem Fall müssen zwischen der Typ-Deklaration und dem Rumpf des Typs mit der Direktive permits die zulässigen Untertypen festgelegt werden (als Komma-getrennte Liste von Typ-Bezeichnern). Für das obige Beispiel ergibt sich daher folgende Typ-Deklaration:

```
public abstract sealed class Shape
        permits Circle, Rectangle, Polygon { ...
```

Somit kann eine versiegelte (abstrakte) Klasse definieren, welche Unterklassen von ihr abgeleitet werden dürfen. Analog kann eine versiegelte Schnittstelle festlegen, welche Klassen als Implementierung erlaubt sind.

Die als zulässig deklarierten Untertypen müssen ihrerseits von dem Obertypen abgeleitet werden. Außerdem muss jeder Untertyp explizit die weitere Steuerung der Vererbungs-Hierarchie auf eine der drei folgenden Arten definieren:

1. Durch die (geschachtelte) Deklaration mit sealed muss nun der Untertyp seinerseits die für ihn zulässigen abgeleiteten Typen festlegen.
2. Durch die Deklaration mit dem neuen Schlüsselwort non-sealed kehrt der Untertyp gewissermaßen in die altbekannte Java-Welt zurück, d. h. es sind nun beliebige und beliebig viele abgeleitete Typen erlaubt.
3. Durch die Deklaration mit dem bekannten Schlüsselwort final (d. h. Ableitungsverbot) endet die versiegelte Vererbungs-Hierarchie mit diesem Untertyp.

Ferner gilt die allgemeine Randbedingung, dass in einer versiegelten Vererbungs-Hierarchie alle beteiligten Typen in einem JPMS-Modul (Java Platform Module System) liegen müssen.

Beispiele:

Schachteln von versiegelten Typen

```
public sealed class Rectangle extends Shape
    permits Square, Parallelogram { ...
```

Öffnen der Versiegelung

```
public non-sealed class Polygon extends Shape { ...

public class Triangle extends Polygon { ...
```

Abschliessen der Vererbungs-Hierarchie

```
public final class Circle extends Shape { ...
```

Zu beachten ist außerdem, dass sowohl Records (record) als auch Aufzählungstypen (enum) zum Abschließen einer versiegelten Vererbungs-Hierarchie verwendet werden können, weil beide Arten von Typen (implizit) final sind.
Mit versiegelten Typen (*Sealed Classes*) und unveränderbaren Daten-Behältern (*Records*) integrieren sich nun alle Java-Typen - also Klassen, Interfaces, Enumerations und Records - zur Beschreibung von Domänen-Objekten in das Konzept der algebraischen Datentypen - also Summentyp und Produkttyp. Die Modellierung solcher fachlichen Objekte ist nach wie vor auch auf abstrakter Ebene (d. h. mit Interfaces oder abstrakten Klassen) möglich, kann aber durch Versiegelung auf die fachlich definierte Vererbungs-Hierarchie eingeschränkt werden. Wie wir noch sehen werden, ist damit die allgemeine Grundlage für das Pattern Matching gelegt.
Insbesondere ist in versiegelten Vererbungs-Hierarchien eine weniger defensive Programmierung möglich, weil der Java-Compiler bereits zur Übersetzungszeit infolge der festgelegten Untertypen die Vollständigkeit von Fallunterscheidungen überwachen kann. Umgekehrt löst der Java-Compiler einen Fehler aus, wenn ein nicht zulässiger Untertyp in eine versiegelte Vererbungs-Hierarchie eingeführt wird, d. h. der Java-Compiler überwacht die fachliche motivierte Vererbung. Die gewünschten fachlichen Einschränkungen der Vererbung erfolgen mit versiegelten Typen außerdem nunmehr deklarativ anstelle von programmatischer Behandlung mittels else- oder default-Zweigen in Fallunterscheidungen.

Jens Seekamp [jens.seekamp@gedoplan.de]
Senior Consultant, Software-Architekt und Dozent mit langjähriger Praxiserfahrung rund um Java und Java EE bei der GEDOPLAN GmbH.

GEDOPLAN
IT Training

Schwerpunkt-Thema bei GEDOPLAN: Quarkus

Quarkus ist ein Open Source Server Framework von Red Hat, das durch einen geringen Speicherverbrauch und eine sehr geringe Startzeit gekennzeichnet ist. Es ist eine gute Alternative zu Spring Boot oder klassischen JEE-Servern.

Quarkus basiert auf den Standards Jakarta EE und MicroProfile, die auch von klassischen Servern wie WildFly oder OpenLiberty implementiert werden. Somit können Basistechniken und Architekturentscheidungen unabhängig von der Zielumgebung genutzt werden. Mit Quarkus erstellen Sie leichtgewichtige Microservices oder Selfcontained Systems.

Wir haben mittlerweile viele offene Schulungen zu dem Thema durchgeführt als auch maßgeschneiderte Firmenschulungen. Ist das Thema Quarkus bei Ihnen aktuell?

Gerne geben wir Ihnen einen Einstieg in dieses Framework mit unserem kostenlosen Vortrag Quarkus reloaded 2.x in Ihrem Hause oder stellen Ihnen in einem Beratungsgespräch unser umfassendes Unterstützungs- und Schulungsangebot zu Quarkus vor. Sprechen Sie uns an!

Unser Geschäftsführer Dirk Weil hat einen Artikel zum Thema Quarkus geschrieben:

"Quarkus – leichtgewichtiges JEE mit Spaß!"
https://gedoplan.de/dirkweil/

Unsere Quarkus-Schulungen im Überblick

Schnelleinstieg in Quarkus
Anwendungsentwicklung mit Quarkus

In diesem Seminar erlernen Sie die Nutzung von Quarkus als Entwicklungsumgebung und Runtime-Framework für Anwendungen auf Basis von Jakarta EE und MicroProfile.

gedoplan.de/kursthemen/jakarta-ee-java-ee/schulung-einfuehrung-quarkus/

Microservices mit Quarkus - kompakt
Jakarta EE mit MicroProfile kombinieren, um Microservices und verteilte Systeme für Quarkus zu entwickeln

Der Fokus dieser Schulung liegt auf auf den Möglichkeiten, JEE-Anwendungen um MicroProfile-Bausteine zu ergänzen, um verteilte Services zu entwickeln.

Mit den MicroProfile-Anteilen Config, Health und Monitoring machen Sie Ihre Services bereit für den Betrieb in Container-Umgebungen wie bspw. Kubernetes.

gedoplan.de/kursthemen/jakarta-ee-java-ee/microservices-mit-quarkus-kompakt/

Workshop: Migration von JEE-Anwendungen zu Quarkus-(Micro-)Services
Umbau bestehender Java-EE/Jakarta-EE-Anwendungen in Microservices auf Basis von Quarkus

In diesem Seminar gehen wir von einer klassischen JEE-Serveranwendung aus - mit den Bestandteilen JPA, EJB und CDI, REST, JSF - und zerlegen sie schrittweise in autonome Services auf Basis von Quarkus. So werden die Unterschiede zwischen Quarkus- zu JEE-Anwendungen sichtbar und die Besonderheiten von Quarkus deutlich und eine schrittweise Migration möglich.

gedoplan.de/kursthemen/jakarta-ee-java-ee/migration-jee-quarkus/

Maßgeschneiderte Firmenschulungen und Schulungsreihen

In letzter Zeit haben wir viele individuelle Firmenschulungen und vermehrt ganze Schulungsreihen für mittelständische und große Unternehmen durchgeführt. Dabei war es wichtig, die Teilnehmer von einem bestimmten Level abzuholen, um in neuen, modernen Sprachen und Tools wie Quarkus oder Docker und Kubernetes fit zu machen. Firmenschulungen sind immer individuell, das bedeutet, dass jede Firmenschulung perfekt geplant werden muss. Für die Schulungsteilnehmer eines Kunden war es beispielsweise die Aufgabe, einen in die Jahre gekommenen Monolithen zu zerschlagen, um diese veraltete Anwendung in neue, performantere Gewänder zu packen. Das bedeutet nicht nur die Schulung einer Programmiersprache, sondern die Durchführung einer ganzen Ausbildungsreihe. Aber auch „nur simple" Migrationen bedeuten immer die Ausbildung eines oder mehrerer Teams mit mehreren Durchführungen, denn alle Mitarbeiter müssen auf einen Kenntnisstand gebracht werden.

Wie gehen wir dabei vor?

Wichtig ist es zu erfahren, welche Kenntnisse die Teilnehmer mitbringen und welche Erfahrungen vorhanden sind. Der Trainer muss wissen, ob er eine homogene oder heterogene Gruppe vor sich hat, um eine optimale Lerngeschwindigkeit zu ermitteln. Im nächsten Schritt werden konkrete Ziele besprochen und dabei ist eine Frage entscheidend: Was und wann muss das Team nach den Schulungen leisten? Unser Tipp: Es sollte in den Vorgesprächen immer ein Teilnehmer aus dem Projektteam dabei sein, nicht nur die verantwortliche Person. Der Trainer muss aus der Praxis kommen und die Sprache oder das Tool selber in einem Softwareprojekt angewandt haben. Es gilt also die perfekte Vorbereitung in Absprache mit dem Kunden.

Die Aufgabe des Trainers

Neben den fachlichen Kenntnissen sind auch didaktische Fähigkeiten von großer Bedeutung. Der Trainer muss neben der Vermittlung der Funktionen in der Lage sein, die Teilnehmer von den neuen Tools zu überzeugen, auch wenn diese Entscheidung für das Tool längst von der Teamleitung gefallen ist. Mit Aussagen wie „Die Funktion kann das (alte) Tool doch genauso gut und bei dem Schritt sogar viel besser!" sind unsere Dozenten häufig konfrontriert. Wir setzen daher nur erfahrene Trainer ein, die über hohe didaktische Fähigkeiten verfügen, um auch diese Teilnehmer von „dem Neuen" überzeugen zu können.

Themen, Tempo und Tiefe

GEDOPLAN erstellt auf Unternehmensziele zugeschnittene Firmenschulungen auch durch unser modulares Schulungssystem, welches stetig erweitert und aktualisiert wird. Die optimale Weiterbildung der Mitarbeiter nach Vorgaben, Anforderungen und Zielen steht im Mittelpunkt.

Unser Schulungskonfigurator

Durch unser modulares Schulungssystem sind wir in der Lage, anhand Ihrer Angaben eine individuelle Schulung zu gestalten.

Unser Konfigurator im Internet bringt Sie leicht zu Ihrer individuellen Firmenschulungsanfrage!

https://gedoplan.de/firmenschulungen/

Einige Schulungsbeispiele

Für einen Kurier-Express-Paket-dienst haben wir die Teilnehmer in dem 2-tägigen Kurs "Neuerungen in Java 8 - 15" (mittlerweile sind wir bei Java 17!) auf den neuesten Java-Stand gebracht. Dieser Kurs war dann die Grundlage für eine Spring Boot-Schulung in englischer Sprache. Mit Apache Kafka wurde die asynchrone Kommunikation zwischen verschiedenen Microservices geschult. Im nächsten Schritt galt es, die Teilnehmer in Git fit zu machen; das Unternehmen plant die Migration von SVN nach Git. Diese 1-tägigen Schulungen werden auf Deutsch als auch in englischer Sprache gehalten.
Für ca. 10 Mitarbeiter eines Softwareherstellers von Buchungssoftware für Veranstaltungen, Videokonferenzen und Desk-Sharing war es das Ziel, die Mitarbeiter für ein Projekt zu qualifizieren, bei demneue Technologien der Jakarta EE zum Einsatz kommen. Entstehen soll eine Art „Container-Anwendung" als Rahmen für weitere zu entwickelnde Anwendungen. Vorab wurden die Grundlagen von Java geschult. Anschließend standen "TypeScript" und "React" im Vordergrund: Technologien, die im Frontend zum Einsatz kommen.

Algebraische Datentypen und Pattern Matching in Java 17, Teil 2

von Jens Seekamp

Was ist Pattern Matching?

Das Pattern Matching (Mustervergleich) ist als ein Verfahren definiert, welches in einer gegebenen Struktur anhand eines vorgegebenen Musters die Bestandteile dieser Struktur identifiziert. Das dafür benötigte Such- bzw. Vergleichsmuster muss dabei mit einer wohldefinierten Syntax bzw. Meta-Symbolen beschrieben werden.
Das ist erst einmal eine schwer verdauliche, abstrakte Definition. Versuchen wir eine erste Annäherung mit der uns allen bekannten Textersetzung in Zeichenketten:

- Wir gehen von der Zeichenkette "Otto Mustermann" als Instanz der Java-Klasse String aus.
- Als Syntax bzw. Meta-Symbol für das Such- bzw. Vergleichsmuster verwenden wir den regulären Ausdruck t{2}.
- Rufen wir nun auf der gegebenen Zeichenkette die Methode replaceAll("t{2}", "nn") auf, so wird der Bestandteil "tt" durch den regulären Ausdruck identifiziert und unsere gewünschte Verarbeitung ersetzt diesen Bestandteil durch die Zeichenkette "nn".
- Im Ergebnis heißt der gute Mann jetzt also "Onno Mustermann".

Konstruktion und Dekonstruktion von Domänen-Objekten

So naheliegend die Analogie der Textersetzung auch ist, trifft sie dennoch nicht den Kern des Pattern Matching. Beim Pattern Matching geht es nämlich vielmehr darum, die Struktur und Bestandteile von fachlichen Domänen-Objekten (z. B. Klasse Circle) zu verarbeiten. Insofern ist das Pattern Matching das duale (in gewisser Weise umgekehrte) Konzept zu den algebraischen Datentypen:

- Alle Java-Typen lassen sich auf Summen- bzw. Produkttypen zurückführen und diese beschreiben die **Konstruktion** von zusammen gesetzten Datenobjekten aus Bestandteilen.
- Jedes zusammengesetzte Java-Objekt kann umgekehrt durch **Dekonstruktion** mittels Pattern Matching in seine Bestandteile zerlegt werden.

Wir konstruieren Java-Objekte durch einen parametrisierten Konstruktor, eine statische Factory-Methode oder über ein (z. B. mit Lombok generiertes) Builder-Pattern:

```
Circle c1 = new Circle(new Point(1.0,1.0), 2.0);

Circle c2 = Circle.of(Point.of(0.0,0.0), 0.5);

Circle c3 = Circle.builder()
            .center(Point.of(1.4,3.5))
            .radius(1.8)
            .build();

Shape s = c1;
```

Wir erkennen, dass es verschiedene, komfortable Mechanismen zur Objekt-Erzeugung in Java gibt. Aber wie gehen wir bisher vor, um gegebene Objekte (z. B. Methoden-Parameter) zu verarbeiten, insbesondere wenn wir auf deren Bestandteile zugreifen müssen:

```
public void print(Shape s) {

  if (s instanceof Circle) {
    Circle c = (Circle) s;
    Point center = c.getCenter();
    Double radius = c.getRadius();
    if ((center != null) && (radius != null)) {
      this.plotter.outputCircle(center, radius);
    }
  }
  else if (s instanceof Rectangle) {
    ...
  }
  else {
    throw new RuntimeException("unknown kind
                                of geometrical shape");
  }
}
```

Gegenüber dem eleganten, schnell verständlichen Source-Code zur Objekt-Erzeugung beispielsweise mit dem Builder-Pattern fallen wir bei der Objekt-Zerlegung gefühlt zurück in die Steinzeit der (imperativen) Programmierung. Und wieviel von solchem "instanceof-cast-if-get-else-throw" Source-Code steckt in unserer fachlichen Datenverarbeitung?

Um dies besser zu machen, benötigen wir eine Syntax bzw. Meta-Symbole zur Zerlegung von Domänen-Objekten in ihre fachlichen Bestandteile. Diese Dekonstruktoren liefern uns dann genau die neue Java-Syntax für das Pattern Matching.
Im Folgenden wird versucht, eine vollständige Systematik für ein solches Pattern Matching in Java anhand von algebraischen Datentypen zu umreißen. Dies vermischt bereits in Java 17 enthaltenes Pattern Matching sowohl mit Preview-Features als auch potenziell denkbaren zukünftigen Sprach-Erweiterungen. Durch diese Darstellung soll das grundlegende Verständnis für das uns neue Programmier-Modell des Pattern Matching befördert und das immense Potenzial verdeutlicht werden.

Pattern Matching für Summentypen

Type- und Guarded-Patterns für instanceof (Java 16)

Das Pattern Matching kann zunächst in Verbindung mit dem Operator instanceof verwendet werden, um insbesondere das fehleranfällige Type-Casting überflüssig zu machen.
Ein Type-Pattern entspricht syntaktisch einer Variablen-Deklaration, d. h. es besteht aus einem Typ-Bezeichner und einem Variablen-Bezeichner. Bei einem **Guarded-Pattern** wird optional mit dem booleschen Operator && eine boolesche Bedingung zur weiteren Einschränkung für den Wertebereich der deklarierten Variablen angehängt. Die Anwendung von Type- und Guarded-Patterns auf die versiegelte Shape-Vererbungs-Hierarchie sieht dann folgendermaßen aus:

```
public void print(Shape s) {

  if (s instanceof Circle c && !c.isEmpty()) {
    // erledigt: Circle c = (Circle) s;
    ...
  else if (s instanceof Rectangle r) {
    ...
  }
  else {
     throw new RuntimeException("unknown kind of geometrical
shape");
  }
}
```

Wie anhand des auskommentierten Source-Code ersichtlich, erledigt das Pattern Matching mit Type-Pattern (und ggf. zusätzlichem Guarded-Pattern) folgende Programmschritte:

- prüfen, ob der Methoden-Parameter s eine Instanz vom Typ Circle referenziert
- wenn ja, dann Deklaration und Initialisierung einer Variablen c vom Typ Circle
- das Pattern Matching scheitert (d. h. die if-Bedingung ist false), falls s den Wert null referenziert oder keine Circle-Instanz repräsentiert
- außerdem wird geprüft, dass das referenzierte Circle-Objekt nicht-leer ist (was auch immer das hier fachlich bedeuten mag)

Typisches indonesisches Essgeschirr

Ein Type-Pattern der Form T t kann erfolgreich auf eine Objekt-Variable s vom Typ S angewendet werden, wenn der Typ S cast-kompatibel zu zum Typ T ist, d. h. das Type-Casting von S auf T löst keine Exception aus. Aus diesem Grund kann ein Type-Pattern auch auf generische Typen wie z. B. collection instanceof List<String> list angewendet werden.
Type- und Guarded-Patterns können in Verbindung mit dem Operator instanceof auch auf beliebige, nicht versiegelte Typen angewendet werden:

```
Object seriesOfThings = List.of(1,2,3);
Boolean isNullOrEmpty =
  (seriesOfThings == null)
  || ((seriesOfThings instanceof String s) && (s.length() == 0))
  || ((seriesOfThings instanceof Collection c) && (c.isEmpty()))
  || ((seriesOfThings instanceof Map m) && (m.size() == 0));
```

Wichtig hervorzuheben ist, dass das Pattern Matching null-safe ist, d. h. ein Pattern "passt" niemals auf null; somit werden die immer wiederkehrenden null-Prüfungen nun überflüssig.

Type- und Guarded-Patterns für switch (Preview in Java 17)

Eine weitere Verwendungs-Möglichkeit des Pattern Matching ergibt sich innerhalb von switch-Ausdrücken bzw. Anweisungen. Auch hier wird die mehrfache Fallunterscheidung mit Type-Patterns (und ggf. ergänzendem Guarded-Pattern) realisiert:

```
public void print(Shape s) {

  switch (s) {
    case Circle c -> this.plotter.outputCircle(c);
    case Rectangle r -> this.plotter.outputRectangle(r);
    case Polygon p -> this.plotter.outputPolygon(p);
  }
}
```

Beim Pattern Matching für switch prüft der Java-Compiler nun die Vollständigkeit (exhaustiveness) der mehrfachen Fallunterscheidung. Wenn nicht alle der zulässigen Untertypen eines versiegelten Typen durch einen case-Zweig abgedeckt sind, dann wird ein Fehler ausgelöst. Umgekehrt brauchen wir als Entwickler keinen default-Zweig mehr zu implementieren, weil der Java-Compiler die vollständige Abdeckung aller möglichen Varianten des versiegelten Typs erkennt.
Nun verstehen wir auch, warum dieses Verhalten des Java-Compiler auch für mehrfache Fallunterscheidungen über einem Aufzählungstyp zutrifft. Ein Aufzählungstyp ist ein spezieller, versiegelter Summentyp, weil alle zulässigen Varianten im Source-Code aufgezählt sind. Somit kann auch hier die Vollständigkeit überwacht werden und ein default-Zweig ist nicht erforderlich:

```
public void processStateOfOrder(OrderState orderState) {

  String mapToOrderCode = switch (orderState) {
    case RECEIVED -> "001";
    case CHECKED -> "002";
    case READY_TO_SHIP -> "008";
    case SENT -> "011";
  };
  ...
```

Auch hier können Type- und Guarded-Patterns in mehrfachen Fallunterscheidungen auf sonstige, nicht versiegelte Typen angewendet werden, wobei dann jedoch wiederum ein default-Zweig angegeben werden muss, weil die Vollständigkeit der Fallunterscheidung nicht vom Java-Compiler geprüft werden kann:

```
Object seriesOfThings = "";
Boolean isNullOrEmpty = switch (seriesOfThings) {
  case null -> true;
  case String s && (s.length() == 0) -> true;
  case Collection c && (c.isEmpty()) -> true;
  case Map m && (m.size() == 0) -> true;
  default -> false;
};
```

Pattern Matching für Produkttypen

Record-Patterns

Die Arbeit an Record-Patterns zur Dekonstruktion von Records erfolgt im JEP 405, welcher bislang keinem geplanten Java-Release zugeordnet ist.
Die Definition eines Record-Pattern ist vergleichsweise einfach, weil die Konstruktion von Records über die in ihrer Deklaration spezifizierte, geordnete Liste von Attributen erfolgt (vgl. Record Person weiter oben). Als Dekonstruktions-Muster verwendet man daher bewusst quasi die Signatur des Konstruktors:

```
Person person = new Person(
                "Silke",
                "Musterfrau",
                LocalDate.of(1984,4,22),
                Gender.FEMALE);

if (object instanceof Person(String f, String s, LocalDate b, Gender g) {
  String description =
    g.salutation() + " " + f + " " + s + " was born " + b;
```

Das Record-Pattern wird im Sinne eines Dekonstruktors wie folgt ausgeführt, wobei auffällt, dass viele sonst eigenständige Programmschritte nun zu einem Verarbeitungsschritt zusammengefasst werden:

1. prüfen, ob die Variable object eine Instanz vom Typ Person referenziert
2. wenn ja, dann werden die Bestandteile (d. h. Komponenten) der Person durch Aufruf der Accessor-Methoden ausgelesen
3. die ausgelesenen Attributwerte werden in die Dekonstruktor-Parameter injiziert
4. das Record-Pattern scheitert, falls object den Wert null referenziert oder keine Person-Instanz repräsentiert oder die Komponenten-Variablen für Vorname, Nachname usw. nicht initialisiert werden können

Die Auswertung von Patterns unterscheidet sich von der uns gewohnten Auswertung von Ausdrücken in Java somit in zweierlei Hinsicht:

- Ein Ausdruck wird von innen nach außen ausgewertet, wohingegen ein Pattern von außen nach innen ausgewertet wird.
- Wenn ein Konstruktor bzw. eine Methode aufgerufen wird, dann sind deren Parameter Eingabewerte, die von außen übergeben werden (In-Parameter, Argument). Wenn ein Dekon-struktor als Pattern verwendet wird, dann sind dessen Parameter Ausgabewerte, die von dem untersuchten Objekt quasi an den Programm-Kontext ausgeliefert werden (Out-Parameter, Binding-Variable).

Unter der Annahme, dass wir in der Shape-Vererbungs-Hierarchie den Untertyp Circle als Record implementiert haben, können wir unsere als Ausgangs-Beispiel skizzierte Methode zum Plotten von geometrischen Formen nun wie folgt implementieren:

```
public void print(Shape s) {

  switch (s) {
    case Circle(Point center, Double radius) ->
      this.plotter.outputCircle(center, radius);
    case Rectangle ...;
    case Polygon ...;
  }
}
```

Durch das Record-Pattern wird die null-safe Zerlegung der Circle-Instanz deklarativ programmiert. Da die Vererbungs-Hierarchie versiegelt ist, überwacht der Java-Compiler die Vollständigkeit der switch-Anweisung.
Oftmals definieren wir verschachtelte Strukturen auf unseren Domänen-Objekten, so beispielsweise diese beiden Produkttypen:

```
Point = Double x Double
Circle = Point x Double
```

Dies führt zu einem entsprechend verschachteltem Konstruktions-Vorgang. Umgekehrt resultiert daraus eine Schachtelung von Record-Patterns für den Dekonstruktions-Vorgang. Da der Java-Compiler mittels Typ-Inferenz die Komponenten-Typen bestimmen kann, können wir dabei auch mit dem Schlüsselwort var anstelle konkreter Typ-Bezeichner arbeiten:

```
Circle c = new Circle(new Point(1.0,1.0), 2.0);

if (s instanceof Circle(Point(var x, var y), Double radius)) { ...
```

Wenn das Pattern Matching in diesem Beispiel funktioniert, dann erhalten wir die drei initialisierten Variablen x / y / radius, um mit ihnen im weiteren Programmablauf zu arbeiten.
Insgesamt sollen die verschiedenen Arten von Patterns in konsistenter Art und Weise miteinander kombinierbar sein. Analog zur Schachtelung der Objekt-Konstruktion (durch verschachtelte Konstruktor-Aufrufe von innen nach außen) ergibt sich eine Schachtelung der Objekt-Dekonstruktion (durch sukzessive Pattern-Auswertung von außen nach innen). Komplexe Objekt-Graphen können damit deklarativ navigiert und zerlegt werden.

Dekonstruktoren für Klassen

Wir erinnern uns, dass Records lediglich eine bestimmte Art von Java-Klassen sind und dass selbstdefinierte Klassen für Domänen-Objekte - genau wie Records - Produkttypen sind. Daher liegt es nahe, dass auch beliebige Domänen-Objekte mittels Pattern Matching in ihre Bestandteile zerlegbar sein sollen.
In einer zukünftigen Java-Version ist daher die Definition von De-

konstruktoren für unsere Domänen-Objekte denkbar (s. [2]). Wie bereits ausgeführt, folgen Dekonstruktoren dem Signatur-Aufbau von Konstruktoren, arbeiten jedoch in der umgekehrten Richtung in dem Sinne, dass sie Out-Parameter mit dem internen Zustand des fachlichen Objektes füllen:

```
public class Article {

 public Article(Integer number, String description, BigDecimal price) {

   this.number = number; // In-Parameter
   ...
 }

 // möglicher Dekonstruktor
 public deconstructor Article(
   Integer number, String description, BigDecimal price) {

   number = this.number; // Out-Parameter
   ...
 }
```

Ein Pattern Matching mittels Dekonstruktor ist dann wiederum in Verbindung mit instanceof oder switch verwendbar. Darüber hinaus soll es ein Match-Statement geben, welches ein Domänen-Objekt mittels Pattern Matching in einem Schritt in seine sämtlichen Attribute (d. h. fachlichen Datenfelder) zerlegt. Dies würde einem simultanen, null-safe Aufruf aller Getter-Methoden entsprechen:

```
Article article = new Article(4711, "iMac", new BigDecimal("1599.00"));

// klassische Getter-Kaskade
Integer number = article.getNumber();
String description = article.getDescription();
BigDecimal price = article.getPrice();
if ((number != null) && (description != null) && (price != null)) {
  ...

// mögliches Match-Statement
Article(var number, var description, var price) = article;
...
```

Dekonstruktoren für Objekt-Sammlungen

Als letztes werfen wir einen Blick auf Objekt-Sammlungen bzw. die Konstruktion von Objekten mittels variabler Argumentlisten. Wir erinnern uns, dass Arrays, Collections und Maps homogene Produkttypen sind und daher auch für diese Art von Objekten deren Dekonstruktion mittels Pattern Matching möglich sein soll (s. [3]).
Für Arrays gibt es in Java eine Literal-Schreibweise, um ad hoc ein Array durch Auflistung seiner Elemente zu definieren. Für Collections und Maps gibt es diese Möglichkeit nicht direkt. Jedoch wurden mit Java 9 statische of()-Methoden für diesen Zweck eingeführt. Wir können daher vereinfachend von solchen Factory-Methoden für die Erzeugung der verschiedenen Arten von Objekt-Sammlungen ausgehen:

```
Integer[] a = {1,2,3,4,5};
==> // Syntax nur für verallgemeinerte Betrachtung
Integer[] a = Array.of(1,2,3,4,5};

List<Integer> l = List.of(1,2,3,4,5};

Map<Integer,String> m = Map.of(
                    Map.entry(1, "one"),
                    Map.entry(2, "two"),
                    Map.entry(3, "three"));
```

In Bezug auf Objekt-Sammlungen sind wir oftmals an deren Länge (d. h. Anzahl der enthaltenen Elemente) und bestimmten Einträgen interessiert. Hierfür können wir zwar schon mit einem generischen Type-Pattern den Typ prüfen und mit einem angehängten Guarded-Pattern auch die Länge kontrollieren, müssen dann jedoch auf althergebrachte Weise einzelne Elemente selektieren:

```
public void processNumbers(Collection<Integer> c) {

  if (c instanceof List<Integer> l && l.size() >= 2) {
    Integer i1 = l.get(0);
    Integer i2 = l.get(1);
    System.out.println("sum of first two numbers = " + (i1 + i2));
}
```

Aus einer verallgemeinerten Factory-Methode of() mit einer variablen Argumentliste ergeben sich folgende Möglichkeiten für das Pattern Matching mittels Dekonstruktoren auf Objekt-Sammlungen:

- Array-Pattern passt auf ein Feld mit genau zwei Integer-Elementen; i1 und i2 referenzieren dann die beiden Integer-Elemente aus dem Feld
  `Array<Integer>.of(Integer i1, Integer i2)`
- List-Pattern passt auf eine Liste mit mindestens zwei Integer-Elementen; i1 und i2 referenzieren dann die beiden ersten Integer-Elemente aus der Liste
  `List<Integer>.of(var i1, var i2, ...)`
- Map-Pattern passt auf ein Verzeichnis mit genau drei Einträgen; k-Variablen referenzieren dann deren Schlüssel und v-Variablen deren Werte
  ```
  Map<Integer,String>.of(
     Map.entry(var k1, var v1), Map.entry(var k2, var v2), _)
  ```
- alternatives Map-Pattern zur Selektion der String-Werte zu ihren Integer-Schlüsseln
  ```
  Map<Integer,String>.of(
     Map.entry(1, String s1), Map.entry(2, String s2), _)
  ```

Dabei dient innerhalb eines solchen Patterns die Zeichenfolge ... als Platzhalter für beliebig viele weitere, aber nicht interessierende Elemente einer Objekt-Sammlung bzw. einer variablen Argumentliste. Analog dient das Zeichen _ als Platzhalter für irgendeinen nicht interessierenden Bestandteil (d. h. Komponente, Attribut oder Element) des untersuchten Objektes (sog. "don't care pattern").

Mit dem skizzierten List-Pattern lässt sich nun die oben gezeigte Methode eleganter implementieren:

```
public void processNumbers(Collection<Integer> c) {

  if (c instanceof List<Integer>.of(var i1, var i2, ...)) {
    System.out.println("sum of first two numbers = " + (i1 + i2));
}
```

So, einen haben wir noch - dann soll es aber auch gut sein. Mit den hier vorgestellten Pattern können wir nun auch die Dekonstruktion eines Records mit variabler Komponenten-Anzahl implementieren (vgl. Record NamedTuple):

- Record-Pattern passt für ein NamedTuple ohne Double-Werte
  NamedTuple(String s)
- Record-Pattern passt für ein NamedTuple mit genau zwei Double-Werten
  NamedTuple(String s, Double d1, Double d2)
- Record-Pattern passt für ein NamedTuple mit mindestens einem Double-Wert
  NamedTuple(var s, var d, ...)

Fazit

In diesem zugegebenermaßen recht langen Artikel wurde versucht, zunächst ein ganz grundlegendes Verständnis für die Konstruktion von Domänen-Objekten im Sinne von Summen- und Produkttypen zu unterbreiten. Es sollte aufgezeigt werden, dass wir uns über Java-Konstruktoren oder Entwurfsmuster (Factory, Builder) schon sehr intensiv mit der Objekterzeugung beschäftigt haben. Umgekehrt haben wir aber keine geeigneten - also null-safe, deklarativen und komfortablen - Sprachmittel für die Objektzerlegung. Hier geben wir uns bislang mit dem schlecht wartbaren, fehleranfälligen "instanceof-cast-if-get-else-throw" Source-Code zufrieden.
Aufgezeigt wurde, dass sich Java auf den Weg gemacht hat, Pattern Matching als neues Programmier-Modell für die Dekonstruktion von Domänen-Objekten einzuführen. Da alle denkbaren Datentypen in Java sich auf algebraische Datentypen zurückführen lassen, ist als dualer Mechanismus das Pattern Matching auf alle denkbaren Domänen-Objekte anwendbar.
Bei der ersten Berührung fällt es uns vielleicht noch schwer, uns in diesem Programmier-Modell zurechtzufinden. Aber wie war es denn bei Java 8 und den Lambda-Ausdrücken? Zuerst musste man es verstehen, dann sich daran durch tägliche Praxis gewöhnen und inzwischen ist es einfach cool, eine elegante Stream-Verarbeitung mittels Lambda-Ausdrücken zu implementieren.
Bekanntlich ist die Zahl 42 die Antwort auf alle Fragen des Universums. Hoffen wir einmal, dass wir nicht bis Java Version 42 warten müssen, um die hier skizzierte vollständige Integration von Pattern Matching und Dekonstruktoren in Java genießen zu dürfen.

Quellen

[1] JEP 361 / 395 / 409 / 394 / 406 / 405 (https://openjdk.java.net/jeps)
[2] Brian Goetz: Deconstruction Patterns for Records and Classes (https://github.com/openjdk/amber-docs/blob/master/eg-drafts/deconstruction-patterns-records-and-classes.md)
[3] Brian Goetz, Gavin Bierman: Pattern Matching in the Java Object Model (https://openjdk.java.net/projects/amber/design-notes/patterns/pattern-match-object-model)

Jens Seekamp [jens.seekamp@gedoplan.de]
Senior Consultant, Software-Architekt und Dozent mit langjähriger Praxiserfahrung rund um Java und Java EE bei der GEDOPLAN GmbH.

Straßenrestaurant in Jakarta

News from the Blog

GEDOPLAN betreibt seit Jahren einen Blog, in dem wir über Neuigkeiten und Tipps rund um Java EE berichten. Regelmäßig informieren unsere Mitarbeiter von ihren Erfahrungen in aktuellen Projekten, geben Tipps, wie kleine Probleme mit Java EE gelöst werden können oder regen Diskussionen um unterschiedliche Lösungsmöglichkeiten an.
Einige der wichtigsten Beiträge möchten wir Ihnen in unserer Rubrik „News from the Blog" in unregelmäßigen Abständen kurz vorstellen. Vielleicht wecken wir Ihr Interesse und Sie besuchen unseren Blog unter javaeeblog.wordpress.com, um sich zu informieren und den gesamten Beitrag zu lesen. Oder, was uns noch mehr freuen würde, um sich an den Diskussionen zu beteiligen.

Schwerpunkt MicroProfile

Dirk Weil geht in mehreren Posts auf die neuesten Entwicklungen zum Thema MicoProfile ein.

Config

Was muss bei der Konfiguration Ihrer Anwendung mit Micro Profile 4 geändert werden und welche Einstellungen können unverändert bleiben?

Health

Die Änderungen in dieser Teilspezifikation umfassen u.a. Startup Checks und Properties für „leere" Liveness und Readiness Checks. Achtung: Die Änderungen bedeuten einen Breaking Change.

Metrics

Mit @SimplyTimed lassen sich die erfassten Daten beim Aufruf auf Aufrufanzahl, Gesamtzeit sowie minimale und maximale Durchlaufzeit beschränken.
@ConcurrentGauge zählt die gelichzeitigen Aufrufe einer Methode.

In seinen Blogs zeigt Dirk Weil jeweils beispielhaft die Umsetzung der neuen Features von MircoProfile.

Quarkus Continuous Testing

Seit der Version 2 unterstützt Quarkus das sog. Continuous Testing. Dabei werden im Deveplopment Mode kontinuierlich die Tests des in der Entwicklung befindlichen Projekts durchgeführt.

Runtime-Modelle für Java EE bzw. Jakarta EE

Entgegen dem Grundgedanken von J2EE, dass der Server die Infrastruktur (Transaktionen, Persistenz, Kommunikation, Isolation etc.) zur Verfügung stellt, die dann von Anwendungen nach dem Deployment genutzt wird, werden aus organisatorischen Gründen viele Server aber zunehmend mit nur einer Anwendung betrieben. Auch daraus resultiert die Meinung, dass damit das Konzept des Application Servers unbrauchbar geworden ist und moderne Serveranwendungen nur mit dedizierten Frameworks wie Spring Boot aufgebaut werden können. Dieser Post zeigt, dass das so nicht stimmt und JEE-Anwendungen sehr wohl in modernen Betriebsumgebungen ihren Platz haben.

GEDOPLAN Java EE Blog
Hier finden sich Neuigkeiten, Ideen und Kommentare rund um Java EE

javaeeblog.wordpress.com

GEDOPLAN
IT Training & IT Consulting

GEDOPLAN IT Training Seit 1998 schulen wir Java, viele Seminare auch in englischer Sprache. Unsere Schulungsleiter sind Java-Experten aus der Praxis, die ihr Wissen in Java-Projekten selbst unter Beweis gestellt haben. Unsere Java-Schulungen beinhalten neben den theoretisch notwendigen Grundlagen auch einen entsprechend hohen Praxisanteil. In unseren Seminaren gehen wir auf aktuelle Problemstellungen des Alltags ein. Damit Sie den bestmöglichen Erfolg erzielen, bieten wir zwei Formen an: In offenen Kursen vermitteln unsere Java-Experten ihr Know-how zu unterschiedlichen, definierten Schwerpunkten. Benötigen Sie Expertenwissen für sehr spezielle Java-Fragen oder -Projekte, führen wir individuelle Gruppen- und Firmenschulungen durch, die wir auf Ihre konkreten Bedürfnisse abstimmen.

GEDOPLAN IT Consulting steht seit vielen Jahren für hochwertiges Consulting in den Java-Technologien. Wir setzen auf offene Standards und Open Source-Produkte. Die Java EE-Plattform ist unsere Basis für die Entwicklung betrieblicher Anwendungen. Plattformen wie WildFly und Liferay führen schnell und sicher zum Ziel.
Ob Neuentwicklung, Migration nach Java EE oder Codereviews: Wir entwickeln IT-Systeme als Komplettpakete, unterstützen unsere Kunden aber auch gerne vor Ort.

GEDOPLAN
Unternehmensberatung und
EDV-Organisation GmbH
Stieghorster Straße 60
33605 Bielefeld
Fon: + 49 521 / 2 08 89 10
Fax: +49 521 / 2 08 89 45
info@gedoplan.de

Geschäftsstelle Berlin:
GEDOPLAN GmbH
UPPER WEST
Kantstraße 164
10623 Berlin

Postadresse Berlin:
GEDOPLAN GmbH
Kurfürstendamm 11
10719 Berlin

+49 30 / 2089 82 630
it-training@gedoplan.de.

Herausgeber: GEDOPLAN Unternehmensberatung und EDV-Organisation GmbH | Stieghorster Straße 60 | 33605 Bielefeld
Verantwortlich: Dipl.-Inform. Dirk Weil | Redaktionsleitung: Ulrich Hake | Fon: 0521 2088910 | E-Mail: info@gedoplan.de
Produktion: networker Medienfabrik GmbH | Nachdruck, auch auszugsweise, nur nach Rücksprache mit der Redaktion.