Dipl.-Kfm.
Friedrich Kiltz

Java Programmiergrundlagen
und
JEE-Web Services mit REST

für die Ausbildungskooperation

Inhalt

1 Kurzvorstellung von Java 5

1.1 Das Wesen von JAVA 5

1.2 Entwicklung von JAVA 5

1.3 Infos über JAVA 6

2 Grundlagen der Sprache Java 7

2.1 Software 7

2.2 Die 1. Applikation 7

2.3 Elemente einer Java-Source 9

2.4 Datentypen 11

2.5 Anweisungen 12

3 OOP in Java 13

3.1 OOP-Grundlagen für Java 13

3.2 Vererbung 14

3.3 Variablen 19

3.4 Einsatz und Stellenwert von Interfaces 20

3.5 Konstruktoren und Destruktoren 21

4 Features 23

4.1 Collections 23

4.2 Fehlerbehandlung mit Exceptions 25

4.3 Grundlagen I/O 27

5 GUI-Applikationen 29

5.1 Allgemeiner Ansatz von JavaFx 29

Herkunft und Entstehung 29

5.2 Basis-Komponenten 29

Applikation 29

Stage 29

Scene 29

Nodes 29

5.3 Hallo-Welt-Applikation 31

5.4 Grundlagen 32

Controls 32

Layout 33

FXML und SceneBuilder 34
5.5 JavaFX und OpenJDK 35
6 Nützliches 37
6.1 Namens-Konventionen 37
6.2 Kommentare u. Anweisungen für JavaDoc 38
6.3 Modifiers (Modifikatoren) 41
6.4 API Specification 42
6.5 Erweiterungen seit Java 1.5 43
Weitere Neuerungen 45
7 Web Services mit REST 46
7.1 Allgemeiner Ansatz von REST 46
Herkunft 46
Beschreibung von REST 47
7.2 JAX-RS mit Jersey 48
Nutzung von Jersey im Tomcat 49
Alternativen 51
Dokumente 52
Application und @ApplicationPath 53
7.3 Request, Response 53
Routing zu einer Java-Methode 54
JAX-RS Injection 55
Content Handler 56
Responses 57
7.4 Client-API 58
JAX-RS Client 58
Weitere REST-Clients 59
7.5 REST API Design 59
URLs 59
Rückgabe mit HTTP-Responsecodes 60
Konventionen zum Benennen von Java-Klassen 61
8 Architektur 62
8.1 Die 5 Ebenen in J2EE 62
8.2 J2EE Core Patterns 64
8.3 MVC - Model-View-Controller 65
9 Links & Quellen 66

9.1 JavaFX 66
9.2 REST 66
9.3 Bücher 66

1 Kurzvorstellung von Java

1.1 Das Wesen von JAVA

Die Vielzahl der Programmiersprachen, von denen wir umgeben sind, entstand dadurch, dass man mit der einen oder anderen Programmiersprache das eine oder andere besser machen oder überhaupt erst realisieren kann.

Um zu wissen, warum ich mich für eine Programmiersprache entscheiden soll, muss ich ihr Wesen kennen. Ich muss den Grund erfahren, warum diese Programmiersprache überhaupt entwickelt wurde. Wenn wir davon ausgehen, dass niemand aus lauter Lust und Langeweile eine Sprache entwickelt, liegt in ihrem Wesen ihre beste Einsatzmöglichkeit.

1991 wurde von SUN Mircosystems die Entwicklung einer Programmiersprache in Auftrag gegeben, die Software für die Steuerung von Konsumelektronikgeräten erstellen sollte. Des Weiteren sollte die Kommunikation der Geräte untereinander möglich sein, sodass der Toaster in der Lage sein sollte, dem Feuermelder mitzuteilen, dass die Rauchwolken, die sich gerade ausbreiten nur von einem alten Stück Weißbrot kommen.

James Gosling erkannte bald, dass diese Aufgabe mit C++ nicht zu lösen war und entwickelte eine neue Programmiersprache mit dem Namen **Oak**. Oak war als Name schon vergeben, sodass eine Umbenennung erfolgen musste: **JAVA**

Diese kurze Reise in die Vergangenheit verdeutlicht das Wesen von JAVA:

- klein
- sicher
- Plattformunabhängig

1994 erkannte man in diesen Vorzügen die hervorragende Eignung von JAVA für das Internet.

1.2 Entwicklung von JAVA

- 1991 bei SUN Microsystems entwickelt.
- Erste Lizenzierung 1995 durch Netscape, dadurch erschien die erste JAVA-Version (1.0.2.) auf dem Markt. Diese Version wird von den meisten Webbrowsern unterstützt.
- Anfang 1997 das erste Release mit der Version 1.1.5. Hauptsächliche Änderung bei den Benutzerschnittstellen und der Ereignisbehandlung
- Ende 1997 wurde die Version 1.2 auf den Markt gebracht. *Bei dieser Version redet man auch von JAVA 2.*

- Oktober 1999 wurde die Version 1.2.2 überarbeitet.
- Mit der Version 1.5 wurden deutliche Neuerungen z.B. im Bereich Generics heraus gebracht.
- Da Oracle seit der Version 1.8. (oder auch Version 8) am Lizenzmodell geschraubt hat ist eine starke Verschiebung zum OpenJDK zu beobachten
- OpenJDK ist zur Zeit bei Version 16, in Produktion sind meist Version 11 und 12.

(Stand: 11/2020)

Da bei dem OpenJDK ein paar Besonderheiten zu berücksichtigen sind, die einem Java-Einsteiger unnötig das Leben erschweren werden wir für die Schulungen die Oracle-Version 1.8 nutzen. In dem Projekt können wir eine aktuelle Version des OpenJDK benutzen.

Weitere Infos zu Version und Download unter
http://www.oracle.com/technetwork/java/javase/downloads/index.html

1.3 Infos über JAVA

Empfehlenswerte Bücher:

- "Java 2 in 21 Tagen" *ISBN: 3-8272-5578-3*
- "Java in a Nutshell" *ISBN: 3-89721-190-4*
- "Java Examples in a Nutshell" *ISBN: 3-89721-112-2*
- "Java lernen mit Eclipse 3. Für Programmieranfänger geeignet." *ISBN: 3898425584*
- "Einstieg in Eclipse 3.5" von Thomas Künneth *ISBN: 3836214288*

Empfehlenswerte Links:

- Die offizielle Seite http://www.oracle.com/technetwork/java/index.html
- Die API Specification (mehr dazu in Grundlagen) http://docs.oracle.com/javase/8/docs/api/
- Handbuch der Java-Programmierung http://www.javabuch.de/download.html
- Java ist auch eine Insel http://openbook.galileocomputing.de/javainsel/

2 Grundlagen der Sprache Java

2.1 Software

Der günstigste Weg Java zu nutzen, ist mit Hilfe des von SUN kostenlos bereitgestellten JDK (Java Software Development Kit). Diese Version kann man bei Oracle herunterladen.

In diesem JDK sind u. a. folgende Programme enthalten:

- java.exe *der Interpreter*
- javac.exe *der Compiler*
- appletviewer.exe *Anzeigeprogramm für Applets*
- jar.exe *für die Erstellung von Paketen (*jar-files)*
- jdb.exe *der Java-Debugger*
- javadoc.exe *Programm zur Erstellung der Dokumentation*

Bei der Installation ist ein besonderes Augenmerk auf folgende Umgebungsvariablen zu legen:

- CLASSPATH *Die Pakete, die eingebunden werden (allen voran "tools.jar")*
- PATH *bitte um das "bin"-Verzeichnis erweitern*
- JAVA_HOME *das Installationsverzeichnis des JDK*

Die Parameter zur Ausführung und Compilierung behandeln wir zu einem späteren Zeitpunkt (bei der Erstellung des 1. Programmes).

Als sehr empfehlenswerte **Open Source** IDE hat sich Eclipse bewährt.

2.2 Die 1. Applikation

Die erste Applikation, in der Datei *Erste.java* abgespeichert, schaut folgendermaßen aus:

```
public class Erste
{
   public static void main( String[] argumente )
   {
      System.out.println( "Es wäre eine haushaltstechnische" +
        " Keuschheit, die sich durch den Gartenschlauch an die " +
        " TV-Serien galaktischer Prägung feucht klemmt.");
   }
}
```

Um die Applikation auszuführen, benötigen wir folgende Schritte:

1. kompilieren mit ***javac Erste.java***

2. ausführen mit ***java Erste***

Man beachte: kompilieren mit Dateiendung (die Datei), ausführen ohne Dateiendung (die Klasse).

Was lernen wir nun aus diesem hutzeligen, trivialen Beispiel?

1. Wir haben eine funktionierende Applikation erstellt, *mit gerade 546 Bytes Größe*.

2. Wir haben eine Klasse und eine Methode erstellt, *JAVA arbeitet nur mit Objekten*.

3. Der Dateiname muss mit dem Namen der Klasse übereinstimmen *(VORSICHT: case-sensitiv)*.

4. Jede Applikation muss eine Methode **main** besitzen, sie ist der Einstiegspunkt für den Interpreter. Main hat immer folgenden Aufbau:

    - **public** damit man von außen auf die Methode zugreifen kann
    - **static** eine statische Methode (Klassenmethode)
    - **void** kein Rückgabewert
    - **main( )** der Methodenname
    - **String** der Typ des Parameters (ein *Array* vom Typ *String*)
    - **argumente** die Variable für den Parameter

5. Mit dem Konstrukt System.out.println() können wir Text ausgeben.

Dabei bedeutet:

- **System**
    - - eine Klasse aus dem Paket *java.lang*
    - - *java.lang* ist eines der wesentlichsten Pakete von JAVA
    - - Klasse kann nicht instanziiert werden (kein *new System()*)
    - - Alle Methoden und Variablen dieser Klasse sind statisch.
- **out.** eine Konstante der Klasse *System* (Standard-Ausgabegerät), die einen *PrintStream* zurückgibt.
- **println()** eine Methode der Klasse *PrintStream*, die die Ausgabe vornimmt

6. Man kann Anweisungen über mehrere Zeilen verteilen.

7. Man kann Zeichenketten nicht über mehrere Zeilen verteilen.

8. Man kann mit einem Dreizeiler die korrekte Installation des JDK überprüfen.

Häufige Problemfälle

1. beim Kompilieren
Erste is an invalid option or argument. Aufruf ohne Dateiendung

2. beim Ausführen
Exception in thread "main" java.lang.NoClassDefFoundError: Erste
- Überprüfung des CLASSPATH

2.3 Elemente einer Java-Source

(aus syntaktischer Sicht)

Identifiers

Identifers (oder auch Bezeichner) sind die Namen für **Variablen**, **Methoden** und **Klassen**.

zum Beispiel:

```
public class Erste
{
   public static void main( String[] argumente )
   {
      System.out.println( "Es wäre eine haushaltstechnische" +
        " Keuschheit, die sich durch den Gartenschlauch an die " +
        " TV-Serien galaktischer Prägung feucht klemmt.");
   }
}
```

Hier sind z. B. **Erste**, **main** und **argumente** Identifiers.

Regeln für Identifiers:

- Jedes Zeichen ist entweder eine Zahl, ein Buchstabe, ein Unterstrich (_) oder ein Währungssymbol ($,¢, £ oder ¥).
- Das erste Zeichen darf keine Zahl sein.
- Ein Identifier darf kein Keywort (s. u.) sein.

Keywords

Keywords (oder Schlüsselworte) sind reservierte Worte, die in Java eine besondere Bedeutung haben.

Eine Liste der wichtigsten Keywords:

abstract	boolean	break	byte	case
catch	char	class	const	continue
default	do	double	else	extends
final	finally	float	for	goto
if	implements	import	instanceof	int
interface	long	native	new	null
package	private	protected	public	return
short	static	strictfp	super	switch
synchronized	this	throw	throws	transient
try	void	volatile	while	

Anmerkung: alle Keywords sind "lowercase". **Abstract** ist kein Keyword!

Literale

Java unterscheidet folgende Literale:

- Zahlenliterale, z. B. **0, 1l, 3.41, 3.0f**
- character literal z. B. **'c'**
- boolean literals **true** und **false**
- String-literal z. B. **"Eine Zeichenkette"**

Kommentare

Java unterscheidet wie auch C

- einzeilige Kommentare: **//** *Rest ist Kommentar*
- mehrzeilige Kommentare: **/*** *ein langer Kommentar* ***/**

Zusätzlich werden noch Kommentare für **Javadoc** (später ausführlich) unterschieden: **/**** *ein langer Kommentar, der in die Dokumentation übernommen wird* ***/**

(mehr dazu im Anhang: JavaDoc)

2.4 Datentypen

Einfache Datentypen

oder auch **primitive Datentypen**

Typ	Inhalt	Standard	Größe
boolean	true / false	false	1 Bit
char	unicode	\u0000	16 Bit
byte	-128 bis 127	0	8 Bit
short	-32.768 bis 32.767	0	16 Bit
int	-2.147.483.648 bis 2.147.483.647	0	32 Bit
long	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807	0	64 Bit
float	1.4E-45 bis 3.4E+38	0.0	32 Bit
double	ganz groß...	0.0	64 Bit

Für jeden primitiven Datentyp gibt es eine entsprechende "Wrapper"-Klasse:

Datentyp Wrapper-Klasse

int:	Integer
short:	Short
long:	Long
byte:	Byte
char:	Character
float:	Float
double:	Double

Referenzdatentypen

Die restlichen Datentypen

- Arrays
- Objekte (s. Kapitel *Java und OOP*)

2.5 Anweisungen

Bedingte Ausführung

Für die bedingten Anweisungen nutzt man **if** oder **case**

```
if   ( logischer Ausdruck )
{
  Anweisungen..
}

switch ( Variable )
{
    case   Wert  :
      Anweisung...
        break;     // sonst führt er den Rest auch noch aus!
    case   Anderer_Wert  :
      Anweisung...
        break;
    default:
      wenn sonst nicht paßt
}
```

Anmerkung: Die Variable muss vom Typ **byte, char, short, int**, **long** oder **String** sein.

Schleifen

for-Schleife

```
for (   Initialisierung  ;   Bedingung  ;   Inkrement   )
{
  Anweisungen...
}
```

while-Schleife

```
while (   Bedingung   )
{
  Anweisungen...
}
```

do...while-Schleife

```
do
{
  Anweisungen...
} while (   Bedingung   );
```

Anmerkung: Die **do...while**-Schleife wird immer mindestens einmal ausgeführt!

3 OOP in Java

3.1 OOP-Grundlagen für Java

Klasse: ein Modell eines Objektes

```
class EineKlasse { .... }
```

Objekt: eine Instanz einer Klasse

```
EineKlasse referenzVariableAufObjekt;       // Deklaration

referenzVariableAufObjekt = new EineKlasse();   // Erzeugung

referenzVariableAufObjekt.eineMethode();    // Kommunikation
```

Eigenschaften: Daten (Variablen) einer Klasse

```
class Kreis
{
    Point mittelpunkt;
    int radius;
    ....
}
```

Methoden: Operationen, die mit einem Objekt durchgeführt werden können

```
class Kreis
{
    Point mittelpunkt;
    int radius;

    public void setRadius( int neu )
    {
        ....
    }
}
```

Subklasse: eine Ableitung einer Klasse und **Superklasse:** Eltern-Klasse einer Subklasse *(s. Vererbung)*

```
class SubKlasse extends SuperKlasse
{
    ...
}
```

Instanzvariable: Eine Variable, die erst in einem Objekt verfügbar ist, und in dem Objekt gekapselt ist (Unterschiedliche Objekte einer Klasse können unterschiedliche Werte für eine Instanzvariable haben).

Klassenvariable: ohne Instanz verfügbar. Für alle Instanzen gleich. *(s. Variablen)*

```
class MerkWerte
{
    // eine Instanzvariable
    private int instanz = 0;

    // die Klassenvariable
    private static int klasse = 0;

}
```

3.2 Vererbung

Ziel dieses Kapitels ist...

- ...den Vorgang der Vererbung in der Programmierung zu kennen
- ...die Gefahren und Möglichkeiten der Vererbung zu erkennen
- ...die Grundlagen der Aggregation kennenzulernen

Die Vererbung in der OOP

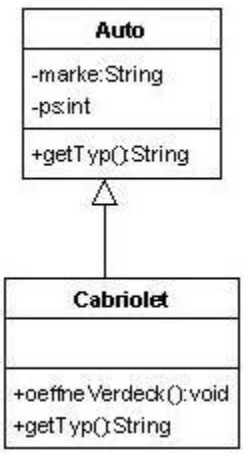
Im täglichen Leben helfen wir uns damit, dass wir Klassen über die Vererbung beschreiben.

Auf die Frage:

"Was ist ein Cabriolet?"

können wir folgende Antwort geben:

*"Ein Cabriolet **ist ein** Auto, bei dem man das Verdeck öffnen kann."*



Anmerkung: In der UML wird die Vererbung durch ein Dreieck bei der Superklasse symbolisiert.

Eine Vererbung stellt immer eine *"ist ein"*-Beziehung dar. Folgende Begriffe sind hierbei Entscheidend:

- **Superklasse** ist die Klasse die vererbt. (hier: Auto)
- **Subklasse** ist die Klasse die erbt. (hier: Cabriolet)
- **Ableitung** ist der Vorgang der Vererbung. (hier: *Cabriolet* ist abgeleitet von *Auto*)
- **Diskriminator** ist das Merkmal, nach dem abgeleitet wird. (hier: *"bei dem man das Verdeck öffnen kann"*)

Wichtige Regeln für die Vererbung

- Alle **Eigenschaften** der Superklasse müssen in der Subklasse enthalten sein.

  Eine Einschränkung des Wertebereiches ist möglich, sollte aber vermieden werden! (Zusicherung)

- Alle **Methoden** der Superklasse müssen in der Subklasse enthalten sein.

  Der Ablauf innerhalb der Methode kann sich verändern (dynamischer Polymorphismus)

- Subklassen können **zusätzliche** Eigenschaften und Methoden erhalten

Die Umsetzung der Vererbung in Java

Bei obigem Beispiel sieht die Klasse Auto folgendermaßen aus:

```
package oop.vererbung;

public class Auto
{
    private int ps;
    private String marke;

    public String getTyp()
    {
        return "Auto";
    }
    public String getMarke()
    {
        return marke;
    }
    public void setMarke(String marke)
    {
        this.marke = marke;
    }
    public int getPs()
    {
        return ps;
    }
    public void setPs(int ps)
    {
        this.ps = ps;
    }
}
```

Die Klasse Cabrio hat folgenden Quelltext:

```
package oop.vererbung;

public class Cabriolet extends Auto {

    public void oeffneVerdeck(){
        // mach irgendwas sinnvolles
    }
    public String getTyp() {
        return "Cabrio";
    }

}
```

Das Schlüsselwort **extends** stellt hier beim Cabriolet die Vererbungsbeziehung zu dem Auto her. Die Methode **oeffneVerdeck()** ist eine Erweiterung gegenüber dem Auto - die Methode **getTyp()** überschreibt hier die Methode aus der Superklasse.

In der Realisierung wurden in der Klasse *Auto* die Zugriffsmethoden mit hinzugefügt. Diese Methoden sind auch in der Klasse *Cabriolet* vorhanden.

Ein Test für diese Klassen könnte folgendermaßen aussehen:

```
Auto a = new Auto();
a.setPs(120);
a.setMarke("Audi");
String infoA = a.getTyp()+": "+
    a.getMarke() +" mit "+a.getPs()+" PS";
System.out.println(infoA);

Cabriolet c = new Cabriolet();
c.setPs(150);
c.setMarke("BMW");
String infoC = c.getTyp()+": "+
    c.getMarke() +" mit "+a.getPs()+" PS";
System.out.println(infoC);
```

Die Anweisung `System.out.println(infoA)` gibt den String *infoA* auf die Console aus.

Was sind die Inhalte von *infoA* und *infoC*?

Mit dem Schlüsselwort **super** greift man auf die Methoden der Superklasse zu. Die Methode *getTyp* des Cabriolets könnten wir z. B. folgendermaßen erweitern:

```
return super.getTyp() + " Cabrio";
```

Mit *super()* ruft man den Konstruktor der Superklasse auf.

Die Mutter aller Klassen: *java.lang.Object*

Alle Klassen in Java sind automatisch von der Klasse *Object* abgeleitet, sofern sie keine andere Superklasse angeben. Somit ist die Superklasse für unsere Klasse *Auto* die Klasse *Object*.

In der API finden wir die Beschreibung zu Object. Unter anderem gibt es dort eine Methode `public String toString()`. Wenn wir uns in der Klasse Object die Realisierung von *toString()* ansehen, schaut sie folgendermaßen aus:

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

…was in unserem obigen Beispiel für das Auto zu folgendem Ergebnis führt:

```
oop.vererbung.Auto@45a877
```

…was nicht wirklich hübsch ist. Die Methode können wir in der Klasse *Auto* überschreiben:

```
public String toString()
{
    return getTyp() + ": " + getMarke()
        +" mit " + getPs() + " PS";
}
```

…und unser Tester würde mit folgender Anweisung

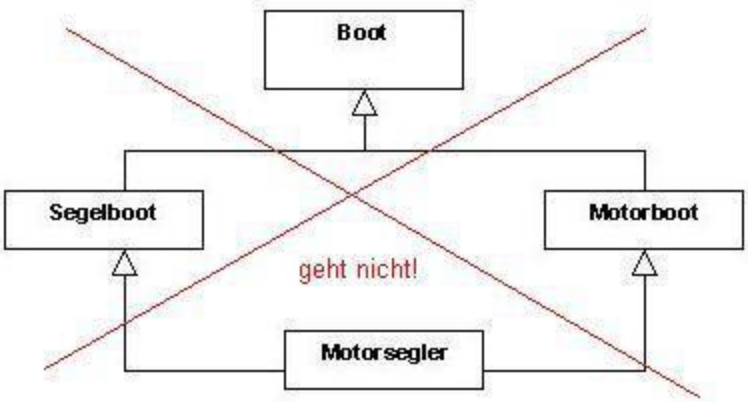
```
Auto a = new Auto();
a.setPs(120);
a.setMarke("Audi");
System.out.println(a.toString());

Cabriolet c = new Cabriolet();
c.setPs(150);
c.setMarke("BMW");
System.out.println(c.toString());
```

zur gleichen Ausgabe kommen wie im obigen Beispiel - nur mit dem Unterschied, dass unsere Info nur einmal zusammengebaut werden muss.

Einfachvererbung

In Java ist nur die Einfachvererbung möglich, dies bedeutet, dass jede Klasse nur eine Superklasse haben kann. Gehen wir von folgender Problematik aus:



Die Klasse *Motorsegler* (Ein Segelboot mit einem Motor) kann nur eine Superklasse haben: *Segelboot* oder *Motorboot*. Die im Diagramm gezeigte Situation ist in Java **nicht realisierbar**.

Overwrite mit dynamischen Polymorphismus

Sofern die Subklasse eine Methode implementiert, die in der Superklasse vorhanden ist und deren Signatur (Name und Parameter) gleich ist spricht man von Overwrite.

Statischer Polymorphismus

Im statischen Polymorphismus ist es möglich, dass mehrere Methoden mit dem gleichen Namen aber unterschiedlicher Signatur existieren. Dies nennt man "Überladen" (overload).

3.3 Variablen

Java unterscheidet zwei Typen von Variablen:

- **Klassenvariablen:** Variablen, die für alle Instanzen der Klasse gleich sind.

Man kann auch sagen: Alle Instanzen nutzen die gleiche Variable.

- **Instanzvariablen:** Variablen, die für jede Instanzen der Klasse neu angelegt wird.

Man kann auch sagen: Alle Instanzen nutzen ihre eigene Variable.

Rein Technisch wird diese Unterscheidung mit dem Modifier static erreicht.

Sinnvoll ist dies insbesondere bei Konstanten oder gemeinsam genutzten Referenzen.

Ein Beispiel

Dieses Beispiel zeigt an zwei Klassen die Nutzung von

1. Instanzvariablen (*instanz*) und
2. Klassenvariablen (*klasse*)

Die entsprechenden Klassen lauten:

- MerkWerte (oop/variable/MerkWerte.java) und
- Rechne (oop/variable/Rechne.java)

Der Output sollte:

```
Objekt1 Klasse: 2
Objekt1 Instanz: 1
```

```
Objekt2 Klasse: 2
Objekt2 Instanz: 2
```

sein.

Daran erkennt man, dass für Objekt 1 die gleiche Variable "iKlasse" genutzt wurde, wie für Objekt 2.

3.4 Einsatz und Stellenwert von Interfaces

Interface

Interfaces (oder Schnittstellen) sind Definitionen von externen Verhalten einer Klasse. Sie definieren eine Menge von Signaturen ohne eine Funktionalität zu implementieren.

Beispiel: Das Interface *Runnable* legt fest, dass es eine Methode *run* gibt.

```
public interface Runnable
{
  public abstract void run();
}
```

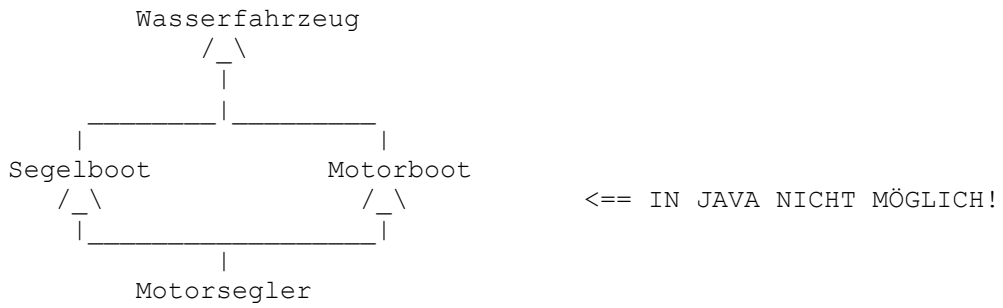
Regeln zu Interfaces:

- ein Interface hat keinerlei Implementation
- alle Methoden sind abstrakt
- alle Methoden sind public
- als Eigenschaften sind nur Konstanten (static final) erlaubt.
- da ein Interface nicht instanziiert werden kann, besitzt es keinen Konstruktor.

Vererbung

Java unterstützt nur die lineare Vererbung, das bedeutet, dass Mehrfachvererbungen nicht möglich sind.

Betrachten wir folgendes Beispiel:



Jedoch kann man in Java eine Klasse durch die Einbindung eines Interfaces *leistungssteigern*, was bedeuten würde, dass ein Motorsegler ein Segelboot (Superklasse) mit zusätzlichen Eigenschaften und Methoden des Motorbootes (Interface) ist.

Beachten Sie, dass das Interface diese Eigenschaften und Methoden nur definiert. Die entsprechende Implementierung erfolgt in "Motorsegler".

Schnittstellen

Ein weitaus wichtigerer Grund für Interfaces ist die Schnittstellenbeschreibung, die ein Interface mit sich bringt. Durch die Definition einer Signatur (Methodenname und Parameter), kann die Kommunikation mit anderen Objekten sichergestellt werden.

3.5 Konstruktoren und Destruktoren

Konstruktor

Der Konstruktor ist eine Methode, die den gleichen Namen hat wie die Klasse. Wenn sie nicht überschrieben wird, wird der Standard-Konstruktor genutzt. Man kann mehrere Konstruktoren nutzen.

Beispiel

```
class Klasse
{
   int i;
   public Klasse()
   {
      i = 0;
   }

   public Klasse( int iNeu )
   {
      i = iNeu;
   }
}
```

Destruktor

Die Methode **finalize()** wird bei einer Klasse immer als letztes ausgeführt. Hier kann man für einen geordneten Abgang sorgen.

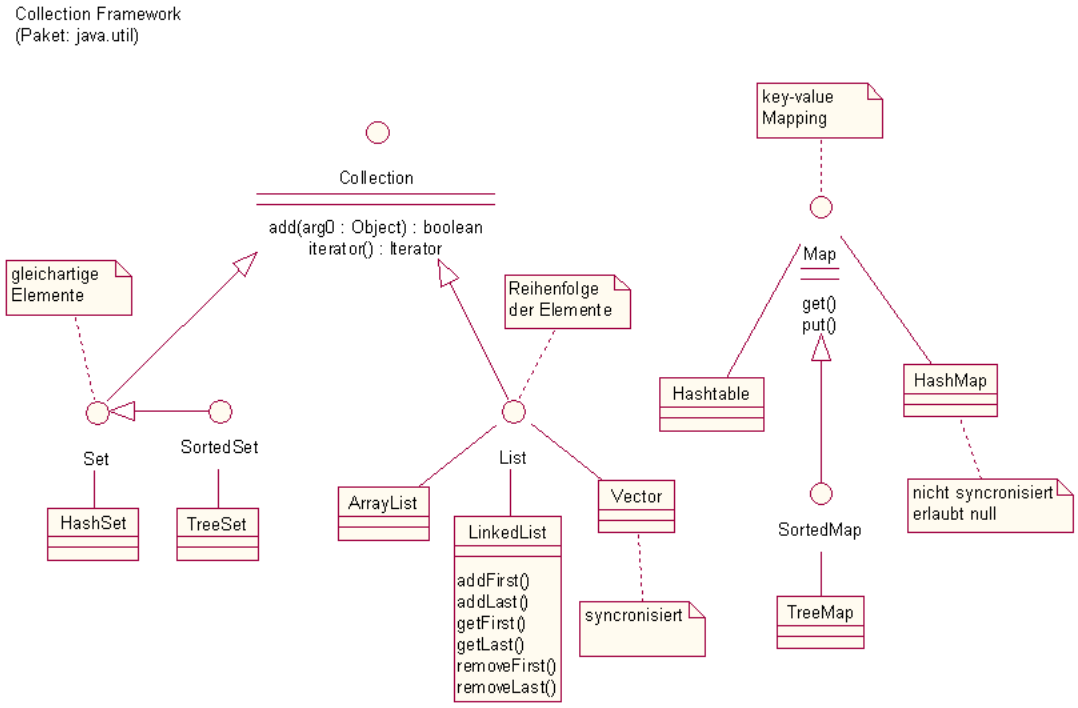
4 Features

4.1 Collections

In diesem Kapitel betrachten wir die Collections (oder auch Container) mit folgenden Schwerpunkten:

- Darstellung des Collection Framework
- Nutzung der richtigen Klassen für die richtige Aufgabe
- Basisoperationen mit Collections
- Sortierung und Comparable

Das Collection Framework



- Das Interface Set ist für gleichartige Inhalte der Sammlung, Dubletten sind nicht erlaubt.
- Bei den Implementationen des Interfaces List werden die Elemente in einer Reihenfolge gehalten, somit sind Dubletten möglich.
- LinkedList wird auch als Stack genutzt.
- Der Vector ist im Gegensatz zu LinkedList und ArrayList synchronisiert. (Dadurch ist die ArrayList ein wenig performanter als der Vector.)
- Maps haben Key-Value Paare, die Hashtable erlaubt keine NULL-Werte und ist synchronisiert.

Nicht vergessen werden sollten die Arrays, die sich ganz klar damit herausheben, dass sie nicht in der Größe veränderbar sind und nur gleichartige Objekte aufnehmen.

Welche Klasse für welche Aufgabe

Grundsätzlich sollten die Objekte vom Typ des Interfaces sein, um eine größere Freiheit bei der Implementation und deren Änderung zu gewährleisten.

```
List einVector = new Vector();
List eineArrayList = new ArrayList();
```

Aufgrund obigem Klassendiagramm und deren Bemerkungen können die Anforderungen der Collection ermittelt und somit die richtige Implementation genutzt werden.

Basisoperationen

- Hinzufügen von Elementen
  - bei einer Implementation der Collection: c.add(Object o)
  - bei einer Implementation der Map: m.put(Object key, Object value)
- Durchlaufen der Elemente
  - bei einer Implementation der Collection:

```
for ( Object o : c )
{
   // tuwas mit o
}
```

  - bei einer Implementation der Map:

```
 for ( Entry<Object, Object> e: m.entrySet() )
{
   Object key = e.getKey();
   Object value = e.getValue();
}
```

Statt dem Datentyp Object kann natürlich auch jeder andere Datentyp (z.B. Kunde, Adresse, String) stehen.

Sortierung und Comparable

Eine Liste l kann mit folgender Anweisung sortiert werden:
Collections.sort(l)

Sofern die Elemente die Schnittstelle Comparable implementieren, funktioniert dies einwandfrei; sofern Comparable nicht implementiert ist, wird eine ClassCastException geworfen.

Mehr Details dazu unter
http://docs.oracle.com/javase/tutorial/collections/interfaces/order.html

Generics

Ab Java 1.5 sollen die Klassen und Interfaces des Collection Frameworks mit Generics genutzt werden. Dabei gibt man bei Erzeugung einer Liste den Typ an, der diese Liste aufnehmen soll. Eine Liste mit Kunden sähe damit folgendermaßen aus:

```
List<Kunde> kundenListe = new ArrayList<Kunde>();
```

Dies stellt sicher, dass die kundenListe nur Instanzen von Kunde aufnehmen kann. Mehr dazu im Anhang unter *Erweiterungen ab Version 5*.

4.2 Fehlerbehandlung mit Exceptions

Eine Exception ist ein Signal von Java, dass eine Ausnahmesituation eingetreten ist.

Zum Beispiel eine Datei nicht gefunden wurde oder man greift auf einen Index eines Arrays zu, der nicht vorhanden ist.

Angenommen ist folgende Klasse:

```
public class EUR2Dollar
{
    public static void main( String[] arg )
    {
        double eur = new Double( arg[0] ).doubleValue();
        double dollar = eur * 1.3596;
        System.out.println( "Dollar: " + dollar );
    }
}
```

Beim Aufruf der Klasse mit einer Zahl als Parameter (*java EUR2Dollar 5.36*) Läuft das Programm und gibt uns den entsprechenden Dollar-Wert aus.

Wenn wir das Programm ohne Parameter starten, greift das Programm auf arg[0] zu und stellt fest, dass dieser Index-Eintrag nicht vorhanden ist.

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException at EUR2Dollar.main(EUR2Dollar.java:5)

Rufen wir das Programm mit einem String als Parameter auf (z. B. "txt"), ist es Java unmöglich diesen String in eine double zu konvertieren:

Exception in thread "main" java.lang.NumberFormatException: txt

Diese Ausnahmen können ganz bewusst in Kauf genommen und als "normale" Ausnahmen behandelt werden. Dies erreicht man durch einen *try - catch – Block.*

```
public class EUR2Dollar  {
    public static void main( String[] arg ) {
        try {
            double eur = new Double( arg[0] ).doubleValue();
            double dollar = eur * 1.3596;
            System.out.println( "dollar: " + dollar);
        }
        catch ( ArrayIndexOutOfBoundsException e ) {
            System.out.println( "Welcher Wert soll berechnet werden?");
        }
        catch ( NumberFormatException e ) {
            System.out.println( "Das war keine gültige Zahl!");
        }
    }
}
```

Hinweise

1. RuntimeExceptions, Errors und deren Subklassen sind unchecked Exception.

Diese müssen nicht abgefangen werden. In der API wird darauf entsprechend hingewiesen:

A method is not required to declare in its throws clause any subclasses of Error that might be thrown during the execution of the method but not caught, since these errors are abnormal conditions that should never occur.

Diese Ausnahmen werden auch **Laufzeitausnahmen** genannt.

Alle anderen Exceptions sind **checked** und müssen behandelt werden. Diese Exceptions werden auch **geprüfte Ausnahmen** genannt. Diese Ausnahmen benutzt man, wenn der Aufrufer den Fehler beheben kann (z.B. durch eine neue korrekte Eingabe), wohingegen die Laufzeitausnahmen auf Programmierfehler in Form von **Vorbedingungsverletzungen** hinweisen. Die ArrayIndexOutOfBoundsException im obigen Beispiel ist z.B. eine solche Laufzeitausnahme. Hier wird **nicht** im Vorfeld geprüft, ob der Index sich in einem gültigen Bereich befindet, sprich ob sich mindestens ein Element im Array befindet.

2. Der Try-Catch-Block kann durch finally (auf alle Fälle) erweitert werden.

Beispiel

Umrechnung eines Euro-Betrages in Dollar
(features/ausnahmen/EUR2Dollar.java)

4.3 Grundlagen I/O

Im Paket **java.io** sind die Funktionalitäten zur Kommunikation mit der Außenwelt zusammengefasst. Die vier Hauptbereiche hierfür sind

- InputStream
- OutputStream
- Reader
- Writer

Streams sind zum Lesen und Schreiben von Byte-Stream, Reader und Writer sind für Zeichen-Streams.

Streams

Am bekanntesten und schon häufig benutzt ist der Standard-Ausgabe-Stream **System.out**. Der Standard-Eingabe-Stream ist **System.in**.

Beispiel:

```
System.out.println( "Weiter mit einer Taste..." );
Try {
     int anzZeichen = System.in.read(  );
}
catch ( IOException e ) {
    System.out.println( "Fehler beim Lesen!");
}
System.out.println( "...Ende" );
```

Reader

Zum Einlesen von Eingaben nutzt man hierfür besser einen gepufferten Zeichen-Stream:

Hier die wesentlichsten Auszüge:

```
// Erzeugung eines gepufferten Readers
BufferedReader eingabe = new BufferedReader(
                         new InputStreamReader( System.in ));

...
// Lesen der Eingabe
name = eingabe.readLine();
```

Komplette Quelle in features/inout/Lese.java

Writer

Im folgenden Beispiel wird ein Passwort über den Standard-Input gelesen (s. Reader) und in einer Datei gespeichert.

Hier die wesentlichsten Auszüge:

```
// Erzeugung eines Datei-Objektes
datei = new File("passwd.txt");
// EIn Filewriter zu dieser Datei
outStream = new FileWriter(datei);
// und schreiben
outStream.write( passwd );
...
// und Stream schließen
outStream.close();
```

Komplette Quelle in features/inout/Schreibe.java

5 GUI-Applikationen

5.1 Allgemeiner Ansatz von JavaFx

Herkunft und Entstehung

Um grafische Anwendungen (GUI) in Java zu erstellen wurde in dem Beginn von Java das AWT (Abstract Window Toolkit) entwickelt. Ab der Version 1.2 (1998) von Java wurde das AWT durch Swing ersetzt. Swing ist nun auch ein wenig in die Jahre gekommen und bietet nicht mehr die Features, die man zur Entwicklung einer modernen Applikation benötigt.

Oracle hatte sich da entschieden nicht mehr Swing weiter zu entwickeln sondern ein komplett neues Framework aufzubauen: JavaFx.

5.2 Basis-Komponenten

Applikation

Eine Fx-Applikation ist von der Klasse javafx.application.Application abgeleitet. Die Klasse bietet den klassischen Lebenszyklus einer Applikation.

1. **launch()** Durch den Aufruf dieser Methode wird der Startvorgang eingeleitet.
2. **init()** (optional) z.B. zur Verarbeitung von Aufrufparametern
3. **start()** (verpflichtend) zum Aufbau der UI
4. **stop()** (optional) für einen geordneten Abgang (eingeleitet durch das "x" oder durch Platform.exit())

Stage

Die Stage (Bühne) wird uns von der Applikation beim Aufruf der Methode start übergeben. Die Stage ist der Top-Level-JavaFX-Container, sozusagen das Root-Element (oder auch Hauptfenster) unserer GUI.

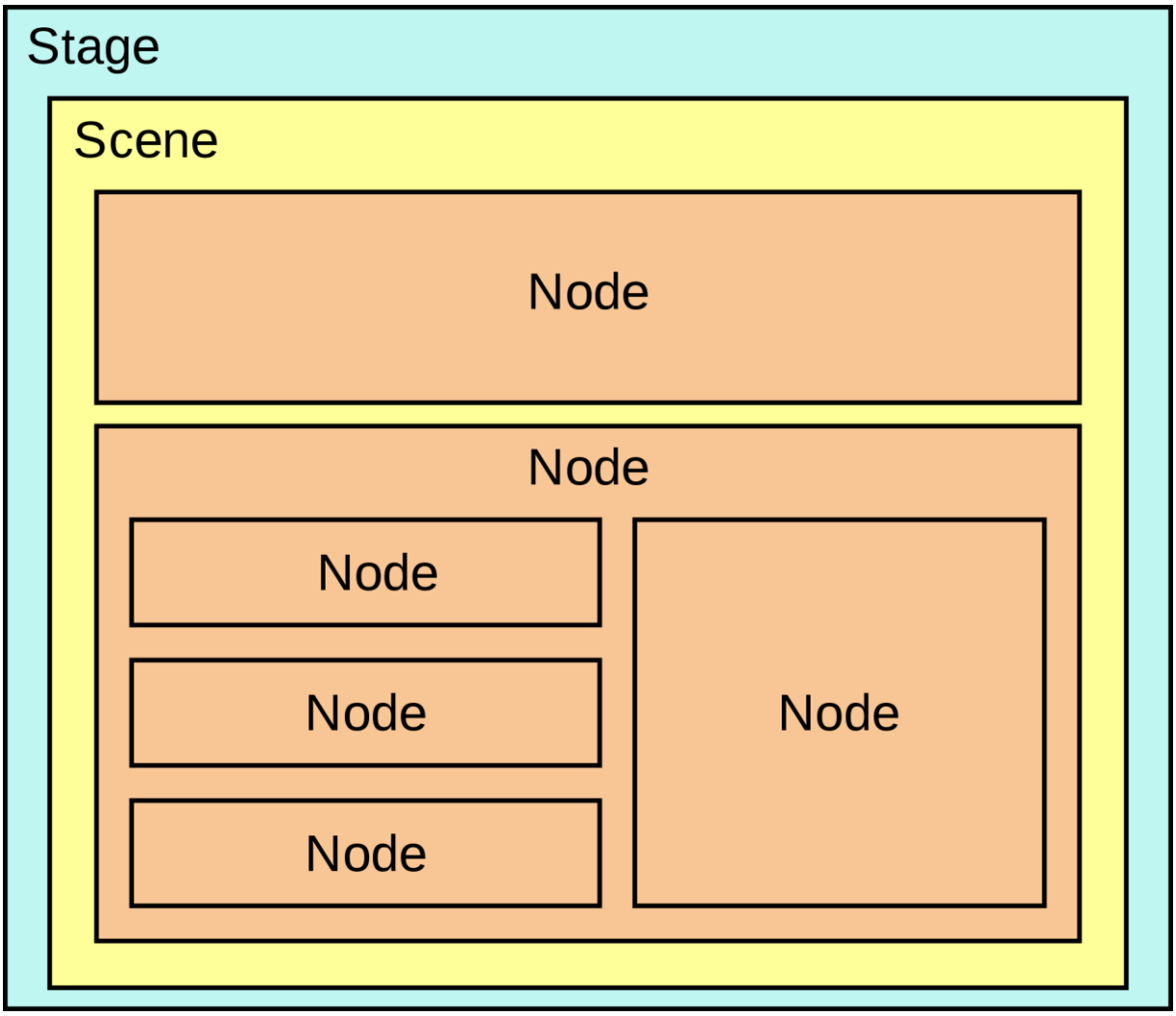
Auf dieser Bühne können wir nun eine oder mehrere Szenen "aufführen".

Scene

Die Scene (Szene), die auf einer Bühne gezeigt wird beinhaltet nun die Elemente, die auf unserem Dialog angezeigt werden sollen. Die Szene ist ein Container, der einen gewurzelten Baum von Dialogkomponenten enthält. Damit ist die Scene ein Verbindungs-Element von der Stage zu ihren Elementen (Nodes).

Nodes

Die Nodes (Knoten, unsere Darsteller) die einer Stage hinzugefügt werden haben immer einen Wurzel-Knoten (gewurzelten Baum) der weitere Knoten enthalten kann. Knoten können weitere Container sein (z.B. für eine Gruppierung oder Layout) oder "Blätter" die das Ende des Astes darstellen und Controls (Button, Label, TextField etc.) sind.



(Quelle: Wikipedia)

5.3 Hallo-Welt-Applikation

```
package de.kiltz.fx.simple01;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HalloWeltApp extends Application {

    @Override
    public void start(Stage primaryStage) {
        Label label = new Label("Eine Kleine Fx-App!");
        StackPane root = new StackPane();
        root.getChildren().add(label);
        Scene scene = new Scene(root, 300, 250);
        primaryStage.setTitle("Kleines Hallo Welt!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }

}
```

Diese kleine unscheinbare Beispiel-Programm enthält die oben genannten Elemente in ihrem normalen Lebensraum.

Die Klasse HalloWeltApp ist von der Klasse javafx.application.Application abgleitet und erhält durch politisch korrekte Erbschleicherei die Methode public void start(Stage primaryStage) die sie überschreiben **muss**. Dieser Methode wird die Stage übergeben, auf die wir nun unsere Szene mit den Darstellern aufbauen können. Die Stage hat ein paar interessante Setter von denen hier nur die wichtigsten vorgestellt sind: Das Setzen des Titels und das setzen der Szene, die hier gespielt werden soll.

In der Szene treten eine Stackpane (als Layout-Container) und ein Label als Control auf.

Weiterhin erhalten wir von der Klasse Application die Methode public static void launch(String… args) die wir hier in der Main-Methode aufrufen.

Problemlos können weitere Bühnen (Dialoge) eröffnet werden. Z.B.:

```
Stage stage = new Stage();
stage.setTitle("Neuer Dialog");
stage.setScene(new Scene(new StackPane(new Label("Nachricht!"))));
stage.show ();
```

5.4 Grundlagen

Controls

Controls sind die Komponenten die Benutzereingaben erlauben, z.B. Button, TextFelder aber auch Labels und SplitPane bzw. TabPane.

Eine kurze Übersicht der wichtigsten Controls:

Control	Verwendung
Label	Eine Komponente aus einem Textfeld und einem beliebigen Dekorations-Node.
Button	Ein einfacher Button.
TextArea	Eine Formularkomponente zur Eingabe oder Editierung längerer Texte.
TextField	Formularkomponente für die einzeilige Texteingabe.
PasswordField	Ein Textfeld, das statt eingegebener Buchstaben ein Symbol darstellt.
CheckBox	Ein Auswahlkasten mit drei möglichen Zuständen (undefined, checked und unchecked).
ChoiceBox	Eine Auswahlliste für die Auswahl eines einzelnen Items aus einer vorzugsweise kurzen Liste. Unterstützt nur Einzelauswahl (SingleSelection).
ComboBox	Eine Auswahlliste, ähnlich ChoiceBox, die für größere Mengen an Objekten
RadioButton	Ein Auswahlknopf für die Einzelauswahl aus einer Gruppe von RadioButtons.
ToggleButton	Ein Auswahlknopf, der in zwei Zuständen existiert, ausgewählt (selected) und nicht ausgewählt. Kann ähnlich dem RadioButton in einer ToggleGroup gruppiert werden.
Slider	Horizontaler oder vertikaler Schieberegler für die Wertauswahl aus einem begrenzten Bereich.
Progressbar	Ein Fortschrittsbalken.
ListView	Standardkomponente zur Anzeige von Listen von Objekten. Erlaubt Einfach- oder Mehrfachselektion und unterstützt Editierung.
MenuBar und Menu	Eine Menüleiste (MenuBar) und die zugehörigen aufklappbaren Menüs (Menu).
MenuButton	Ein Button, der bei einem Klick ein Menü anzeigt.
MenuItem	Ein Menüeintrag, der bevorzugte Inhalt von Menüs. Konkrete Subklassen mit Zusatzfunktionen sind verfügbar (CheckMenuItem, RadioMenuItem, CustomMenuItem).
ContextMenu	Ein Pop-up-Fenster mit Menüeinträgen.
ScrollBar	Ein horizontaler oder vertikaler Rollbalken.
ScrollPane	Komponente, der eine Viewkomponente übergeben wird, mit horizontalem und vertikalem ScrollBar, der bei Bedarf angezeigt wird.
Separator	Eine horizontale oder vertikale Trennlinie.
SplitPane	Eine horizontal oder vertikal ausgerichtete Komponente mit zwei oder mehr Unterkomponenten und verschiebbaren Stegen zwischen diesen.
TabPane, Tab	Eine Komponente (TabPane), die jeweils eine Registerkarte (Tab) aus einer Gruppen von Tabs anzeigt.
TitledPane	Eine Komponente mit einem Titel, deren Inhalt ein- oder ausgeklappt werden

Control	Verwendung
	kann.
ToolBar	Horizontale oder vertikale Werkzeugleiste
ToolTip	Ein Pop-up, das häufig als Information angezeigt wird, wenn man den Mauszeiger über eine Komponente führt.
TableView, TableColumn, TableCell	Eine Tabellenkomponente, die aus einer Reihe von Spalten (TableColumns) besteht. Einzelne Zellen werden als TableCell dargestellt.
TreeView, TreeItem	Eine Baumkomponente (TreeView) zur Anzeige von hierarchisch geordneten Baumknoten (TreeItems).

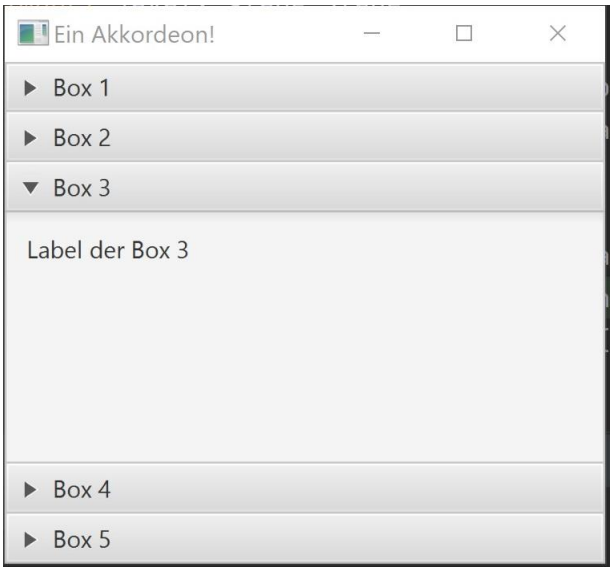
(aus Anton Epple: JavaFX 8)

Layout

Das Layout hat die Anforderung, das es plattformübergreifend gut aussehen soll. Da die einzelnen Komponenten auf unterschiedlichen Betriebssystemen unterschiedliche Größen haben (z.B. Default-Höhe eines Textfeldes ist unter Windows 11 Pixel und Linux 13 Pixel) basieren die meisten Layoutmanager darauf die einzelnen Controls in Relation zueinander anzuordnen.

HBox und **VBox** ordnen die Elemente nebeneinander bzw. untereinander an.

Das **Accordion** lässt Elemente mit Überschriften einklappbar sein.



Mit der SplitPane kann man in der Größe veränderbare Bereiche erstellen und mit der TabPane erzeugt man das was man sich darunter vorstellt:

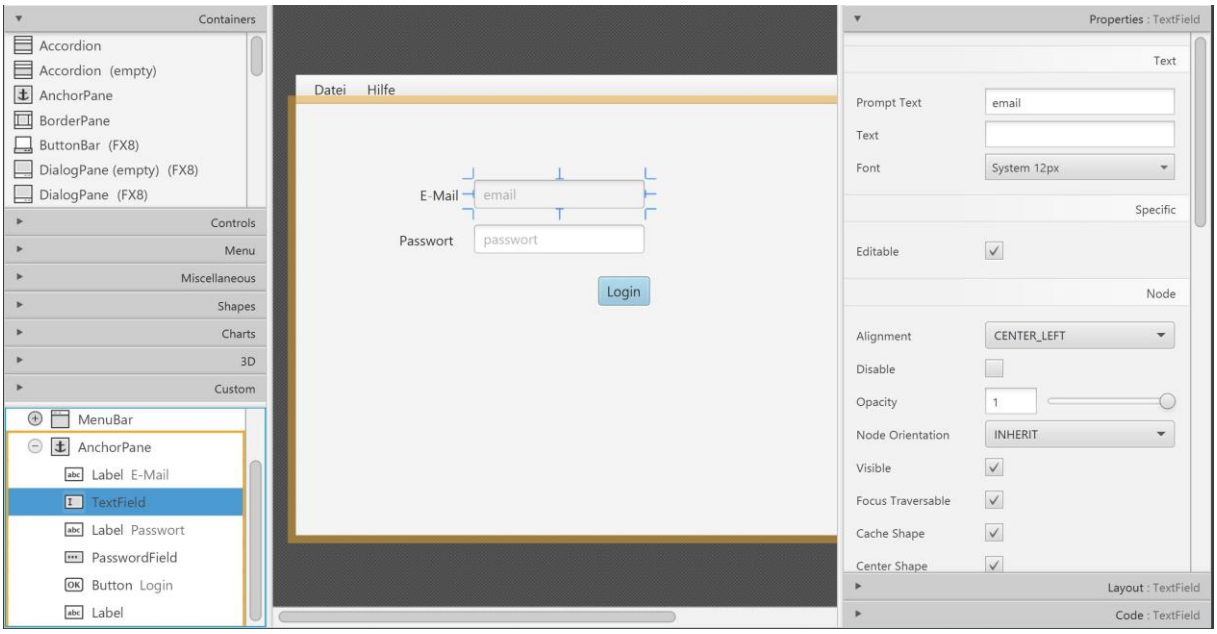
Weitere wichtige Varianten sind:

AnchorPane	„Verankert“ Controls an einem bestimmten Rand oder Control und definiert dazu Abstände.
BorderPane	Bietet 5 Positionen (Top, Bottom, Left, Right, Center)
FlowPane	Ordnet die Elemente hintereinander an.
GridPane	Unterteilt den Anzeigebereich in eine Tabelle mit Abständen und Zellen und Spalten.

FXML und SceneBuilder

Eine wichtige Anforderung für FXML ist die Trennung zwischen der Ansicht (View) und der Funktionalität (Controller). Zu diesem Zwecke ist es möglich mit einer XML-Datei das Layout zu definieren und in einer anderen Komponente (dem Controller) auf die in der XML-Datei definierten Controls zuzugreifen und auf deren Ereignisse zu reagieren.

Diese XML-Dateien unterliegen einem bestimmten Format, das FXML genannt wird. Diese Dateien könne einfach mit einer GUI-Applikation, dem SceneBuilder, erzeugt werden. Hier kann man per Drag & Drop die Dialoge gestalten:



5.5 JavaFX und OpenJDK

Im OpenJDK ab Version 9 ist FX nicht mehr enthalten und muss als Modul in die Applikation mit eingebunden werden. In der Gradle-Umgebung gibt es hierfür Plugins, die uns das Leben erleichtern kann.

```
plugins {
    id 'application'
    id 'org.openjfx.javafxplugin' version '0.0.8'
    id 'org.beryx.jlink' version '2.12.0'
}

sourceCompatibility = 1.12

repositories {
    mavenCentral()
}

javafx {
    version = "11"
    modules = [ 'javafx.controls', 'javafx.fxml' ]
}

mainClassName = "de.kiltz.seminar.fx.MainApp"

jlink {
    options = ['--strip-debug', '--compress', '2', '--no-header-files',
           '--no-man-pages']
    launcher {
        name = 'fxBasics'
    }
}
```

(die `build.gradle`)

Hilfreiche Targets von Gradle:

- **run** : startet die Applikation, die per `mainClassName` in der `build.gradle` eingetragen ist.
- **assemble** : Zur Ausführung in der IDE

6 Nützliches

6.1 Namens-Konventionen

Zur Verbesserung der Lesbarkeit von Java-Quelltexten werden von Java folgende Namens-Konventionen empfohlen:

Pakete

Um die Einheitlichkeit von Paketen sicherzustellen, wird empfohlen, die Paketnamen mit der umgedrehten URL darzustellen. Alle Worte werden in Kleinbuchstaben geschrieben.

z.B.

- com.sun.eng
- edu.cmu.cs.bovik.cheese
- de.kiltz.kurs.grundlagen

Klassen

Klassen-Namen beginnen mit einem Großbuchstaben. Bei zusammengesetzten Worten beginnt jedes Wort mit einem Großbuchstaben.

Klassennamen sollten Hauptwörter sein

- class String
- class Test
- class TestTheWest

Interface

s. Klassen nur keine Hauptwörter, sondern Adjektive

- interface Runnable
- interface Fahrbar

Methoden

Methoden beginnen mit einem Kleinbuchstaben. Bei zusammengesetzten Worten beginnt jedes weitere Wort mit einem Großbuchstaben.

Methoden-Namen sollten Verben sein.

- run();
- gibAus();
- holGehalt();

Variablen

Variablen beginnen mit einem Kleinbuchstaben. Bei zusammengesetzten Worten beginnt jedes weitere Wort mit einem Großbuchstaben.

Die Namen sollten kurz und treffend sein und nicht mit "_" oder "&" beginnen.

i, j, k, m für Laufvariablen/temporäre Integers
c, d, e für temp. Characters

- int i;
- char c;
- String dasWasZuSagenIst
- int anzahlKinder
- int anzahlAnerkannterKinder

Konstanten

Konstanten bestehen komplett aus Grossbuchstaben. Bei zusammengesetzten Worten beginnt jedes weitere Wort mit einem Unterstrich.

- static final int PORT = 5678;
- static final int MAX_LAENGE = 50;

Für weitere Informationen zu den Code Conventions siehe https://www.oracle.com/java/technologies/javase/codeconventions-contents.html.

6.2 Kommentare u. Anweisungen für JavaDoc

JavaDoc - Das Programm

Mit dem Programm **javadoc** kann eine Dokumentation eines Projektes erstellt werden, die aus den Kommentaren und Deklarationen unserer Sourcen generiert wird.

Ein Aufruf des Programms **javadoc** kann z. B. so aussehen:

```
javadoc
    -overview ../overview.html
    -locale de_DE
    -author
    -version -d .
    -classpath ..\..;d:\webserver\jswdk\src
    -windowtitle Termin
    -link http://http://java.sun.com/j2se/1.4/docs/api
    -use
    -private
    @pakete
```

Der Inhalt der Datei **pakete** (s. @pakete) könnte folgenden Inhalt haben:

```
paket1.paket2.Klasse.java
paket1.paket3
paket4.paket5
```

JavaDoc - Kommentare

Beachte: Alle Kommentare werden in HTML geschrieben!

externe Dateien

Projektbeschreibung: overview.html (im Stamm-Verzeichnis)
erlaubte Tags: @see, {@link}, @since

Paketbeschreibung: package.html (in jedem Paket-Verzeichnis)
erlaubte Tags: @see, {@link}, @since, @deprecated

Im Quelltext

für die **Klasse** (bzw. **Interface**)
erlaubte Tags: @see, {@link}, @since, @deprecated, @author, @version

Zwischen dem Kommentar und der Klassendeklaration darf keine Anweisung (z. B. import) stehen!

```
/**
 * Eine Klasse zur Herstellung der Verbindung zu einem MySQL-Server.
 *
 * Die Klasse nutzt den MySQL-Connector von MM
 *
 * @author  Friedrich Kiltz
 * @version 1.0
 */
class Verbindung
{
   ...
}
```

vor **Klassen-** und **Instanzvariablen**
erlaubte Tags: @see, {@link}, @since, @deprecated, @serial, @serialField

```
/**
 * Port zum MySQL-Server
 */
static int port = 3306;
```

Methoden und **Konstruktoren**
erlaubte Tags: @see, {@link}, @since, @deprecated, @param, @return @throws (@exception), @serialData

```
/**
 * Sendet ein select-Statement ab und gibt ein Resulset zurück
 *
 * @param     sql das SQl-Stament
 * @return    rs das Resultset
 * @see       java.sql.Statement#executeQuery()
 */
public ResultSet holSatz(String sql)
{
   ...
}
```

JavaDoc - die wichtigsten Tags

@author: Anzeige des Authors

{ @docRoot }: relativer Pfad zu den generierten Doku-Seiten

@deprecated: um eine Veralterung zu kennzeichnen

@exception oder auch @throws: Darstellung der Exceptions, die die Methode wirft.

{ @link }: Einfügen eines Links. z. B.: Weitere Infos unter {@link paket.Klasse#Methode(int i) LinkText}

@param: Darstellung der Parameter

@return: Angabe des Return-Wertes der Methode

@see: wie @link, muß aber am Anfang der Zeile stehen

@serial: gilt seit der Versionsnummer

@version: Ausgabe der Versionsnummer

Die komplette Anleitung finden Sie unter http://docs.oracle.com/javase/6/docs/technotes/guides/javadoc/index.html

6.3 Modifiers (Modifikatoren)

(Auszug)

Zugriffs-Modifikatoren

- **public** Public Variablen, Methoden und Klassen können von allen Java-Programmen ohne Einschränkung genutzt werden.

- (ohne Angabe eines Modifiers!) friendly Inner-Classes, Methoden und Variablen sind innerhalb eines Paketes.

- **protected** protected Inner-Classes, Methoden und Variablen sind innerhalb eines Paketes, sowie in Ableitungen der Klassen sichtbar.

- **private** Top-Level-Klassen werden nicht als private deklariert! private Methoden oder Variablen sind nur in der Instanz einer Klasse sichtbar!

Andere Modifikatoren

- **final**(für: Klassen, Methoden und Variablen) Auswirkungen:

    - Von einer finalen **Klasse** können keine Ableitungen erstellt werden.

    - Eine finale **Variable** kann nach der Initialisierung nicht mehr verändert werden (Konstante).

    - Finale **Methoden** können nicht überschrieben werden.

- **abstract** (für Klassen und Methoden) Abstrakte Klassen sind als "Ableitungsvorgabe" für Subklassen bestimmt, ohne daß sie selbst referenziert werden sollen.(Pflanze -> (Baum, Blume, ..) Abstrakte Methoden haben keinen Body, sie sind lediglich eine "Signaturdefinition". *Nur abstrakte Klassen können abstrakte Methoden besitzen!*

- **static** (für Variablen und Methoden) Statische Variablen sind für alle Instanzen einer Klasse gleich (Klassen-Variable). Statische Methoden (wie main) können nicht auf nicht-statische Variablen einer Klasse zugreifen.

6.4 API Specification

Alle Klassen von Java sind in der **API Specification** dokumentiert. Sie ist im Internet unter http://docs.oracle.com/javase/7/docs/api/index.html zu finden.

API: Application Programming Interface

Beispiel: *Klasse Integer*

Einzelne Klassen sind in den meisten Fällen nach folgendem Schema beschrieben:

Name der Klasse und Angabe in welchem Paket sie zu finden
ist ganz wichtig für Import!
(java.lang)

Definition, Ableitungen und Implementierungen
Def.: *public final class **Integer***
extends Number
implements Comparable

Dann folgt eine kurze Beschreibung der Klasse, evtl. mit Beispiel für die Nutzung.

Field Summary, die statischen Felder, die für diese Klasse existieren
MAX_VALUE, MIN_VALUE, TYPE

Constructor Summary, die einzelnen Constructors, die möglich sind
mit "int" oder "String"

Method Summary, alle Methoden, die in dieser Klasse implementiert wurden.
byteValue, compareTo, ...

Evtl. Methoden, die durch korrekte Erbschleicherei nutzbar sind
clone, finalize,... von Object (Superklasse von *Number*)

... und deren ausführliche Beschreibung.

6.5 Erweiterungen seit Java 1.5

Mit der Version 1.5 wurden neue Sprachkonstrukte in Java eingefügt, die ich im Folgenden anspreche.

Auszug aus *Schneller Tiger oder müdes Kätzchen?* vom Marc Steger, veröffentlicht in der Java-Spektrum.

Neue for-Schleifenvariante

Die neue for-Schleifenvariante ermöglicht die Iteration über ein Array oder eine Collection ohne Definition eines Schleifen- index oder Iterators. Es muss lediglich eine Variable definiert werden, die je Schleifendurchlauf den aktuellen Wert des Arrays bzw. der Collection annimmt:

```
List aList = ...;
for(Object value: aList) {
// Verarbeitung von value;
}
```

Autoboxing und Unboxing

Unter Autoboxing und Unboxing versteht man die automatische Umwandlung von primitiven Java-Datentypen in ihre Wrapper-Objekte und umgekehrt. In Java 5 ist damit folgender Code möglich:

```
Integer objVal = 4711;
int intVal = objVal;
```

Enums

Mit Java 5 führt Sun den Aufzählungstyp enum ein. Eine Enum-Klasse wird dabei ähnlich einer normalen Klasse definiert. Anstatt class wird jedoch enum verwendet:

```
public enum Season
{
    SPRING, SUMMER, AUTUMN, WINTER;
}
```

Ein Zugriff auf die einzelnen Werte erfolgt dann wie einer auf statische Konstanten:

```
Season aSeason = Season.SUMMER;
```

Dabei erzeugt der Compiler im Hintergrund nach dem ENUM-Pattern folgende Klasse:

```
public class Season {
    public static final Season SPRING = new Season("SPRING");
    public static final Season SUMMER = new Season("SUMMER");
    public static final Season AUTUMN = new Season("AUTUMN");
    public static final Season WINTER = new Season("WINTER");

    private final String name;

    private Season(String pName) { name = pName; }
    public String toString() { return name; }
}
```

VarArgs

VarArgs sind eine neue Möglichkeit, Methoden mit einer variablen Anzahl von Übergabeparametern zu definieren. Dazu wird dem Datentyp des letzten Parameters der Parameterliste einfach nur "…" angefügt:

```
public int sum(int... pNumbers)
{
    int lSum = 0;
    int lCount = pNumbers.length;
    for(int lI=0; lI<lCount; lI++)
    {
        lSum += pNumbers[lI];
    }
    return lSum;
}
```

Die Methode sum() kann somit beliebig viele int-Werte aufsummieren und als Ergebnis zurückliefern.

Folgender Aufruf ist dabei möglich:

```
int lSum1 = sum(1, 2);
int lSum2 = sum(1, 10, 100, 1000, 10000);
```

printf

Keine Spracherweiterung an sich, sondern eine neue Bibliotheksfunktion ist die neue Ausgabefunktion printf(String format, Object… args), die sich der neuen Klasse java.util.Formatter bedient und der gleichnamigen C-Funktion nachempfunden ist. Erst die neuen VarArgs der J2SE 5 haben die Methode in dieser Form ermöglicht.

```
int lLow = 0;
int lUp = 99;
System.out.printf("Der Wert muss zischen '%d' und '%d' liegen.\n",
lLow, lUp);
```

Generics

Generics sind eine Erweiterung des Java-Typsystems. Neben den primitiven Datentypen (int, long, …) und normalen Typen, sprich Klassen und Interfaces, gibt es nun generische Typen. Generische Typen sind Klassen oder Interfaces, die eine oder mehrere Typ-Variablen besitzen. Wird bei der Nutzung dieser Klassen ein konkreter Typ angegeben, ermöglicht dies u. a. eine Typprüfung zum Übersetzungszeitpunkt durch den Compiler.

```
List<Integer> intList = new ArrayList<Integer>();
List<String> strList = new ArrayList<String>();
intList.add(10);
strList.add("Hallo");
Integer number = intList.get(0);
String text = strList.get(0);
```

Weitere Neuerungen

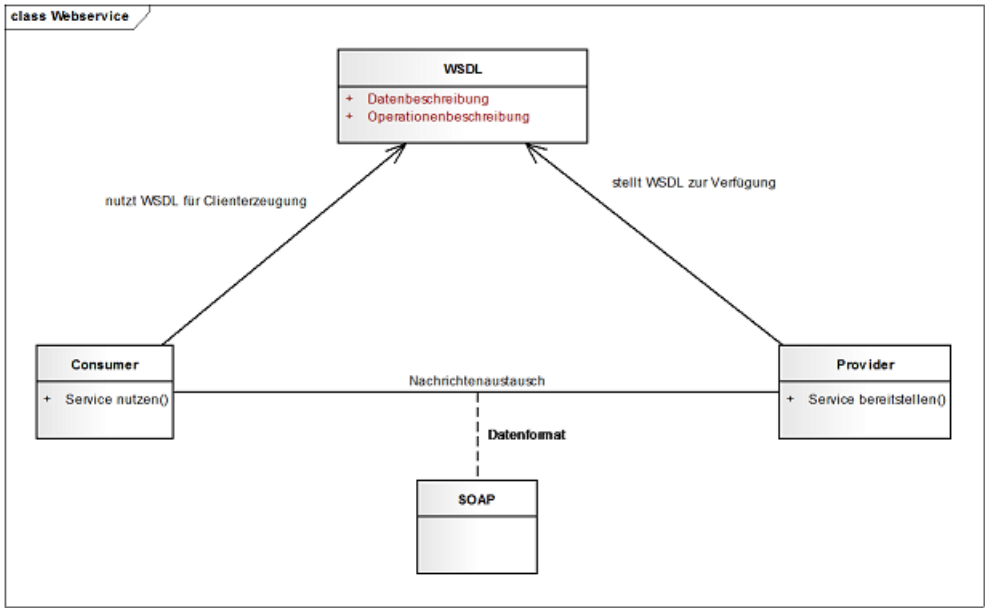
Weitere Neuerung wie Streams, Lambdas etc. sind mehr Fortgeschrittene Themen, die gerne in einem Aufbau-Seminar behandelt werden können.

7 Web Services mit REST

7.1 Allgemeiner Ansatz von REST

Herkunft

Web Services im engeren Sinne nutzen eine WSDL zur Beschreibung der Services mit ihren Datentypen und SOAP-Nachrichten zum Transport der eigentlichen Aufrufe. Dabei können die unterschiedlichsten Protokolle (http(s), ftp, SMTP etc.) genutzt werden.



Die Vorteile dieses Ansatzes stecken in der sehr aussagekräftigen WSDL die eine starke Entkopplung von Provider und Consumer ermöglicht. Außerdem haben wir durch die SOAPNachricht die Möglichkeit im Header zusätzliche Informationen (Security, Adressing etc.) mit zu übertragen. Die unterschiedlichen Protokolle ermöglichen uns auf die Natur der Daten einzugehen.

Roy Fielding stelle diese Basis für eine Reihe von Anwendungsfällen in Frage. Da doch viele Web Services nur mit HTTP(S) arbeiten und Consumer und Provider sich kennen und austauschen können und es wahrlich performantere Möglichkeiten als XML gibt um Daten auszutauschen könnte man die Web Services vereinfachen.

In seiner Dissertation im Jahre 2000 veröffentlichte Fielding den REST-Architekturstil, wobei REST für Representational State Transfer steht.

> *Die Bezeichnung „Representational State Transfer" soll den Übergang vom aktuellen Zustand zum nächsten Zustand (state) einer Applikation verbildlichen. Dieser Zustandsübergang erfolgt durch den Transfer der Daten, die den nächsten Zustand repräsentieren.*

(aus: https://de.wikipedia.org/wiki/Representational_State_Transfer)

Beschreibung von REST

REST lehnt sich an das Prinzip des WWW an und unterstützt nur das Kommunikationsmuster Request/Response. Ein wichtiges Prinzip ist die eindeutige Adressierbarkeit jeder Resource durch einen URI. Hierbei kann der Request per GET oder POST erfolgen und entweder per Query-String oder POST einfache Datentypen, HTML-, XML- oder Binärdaten übertragen. Der Response gibt nur die Nutzdaten zurück. Je nach Anforderung durch den Client kann die Rückgabe in einer unterschiedlichen Repräsentation geschehen, z.B. in HTML, XML, JPG etc. Neben den Zugriffsmethoden POST und GET können die beiden anderen HTTP-Methoden PUT und DELETE für die Änderung oder Löschung von Resourcen benutzt werden.

Im Vergleich zu den Basisfunktionalitäten der Persistenz CRUD (Create, Read, Update und Delete) verhalten sich die HTTP-Methoden folgendermaßen:

GET	Abrufen von Informationen, wie z.B. die Liste der Artikel, einen Kunden, den Status einer Lieferung. Die Daten können eventuell gecached werden (analog zum Read aus CRUD). `GET /kunden/k123` GET-Aufrufe sind nur lesend und idempotent.

POST	Erzeugung einer neuer Resource, wie z.B. der Position einer Bestellung, Bemerkungen zu einem Kunden (analog zum Create aus CRUD).
	
	`POST /bestellungen/0815`
	`<position nr="1" artnr="123" menge="5"/>`
	
	POST-Aufrufe sind **nicht** idempotent.

PUT	Änderung einer Resource mit einer bestimmten ID, wie z.B. den Lieferanten l321 (analog zum Update aus CRUD).
	
	`PUT /lieferanten/l321`
	`<name>Chateau Certan de May</name> <land>Frankreich</land>`
	
	PUT-Aufrufe sind idempotent.

DELETE	Löschen einer Resource, wie z.B. Löschen des Kunden mit der Kd-Nr.: k123 (analog zum Delete aus CRUD). `DELETE /kunden/k123` DELETE-Aufrufe sind idempotent.

Die HTTP-Methoden TRACE, OPTIONS und HEAD werden in der Praxis kaum genutzt.

REST ist genau wie HTTP zustandslos, d.h., jeder Zugriff steht für sich allein. Sessions sollten nur auf der Clientseite verwaltet werden. Dies hat den Nachteil, dass die Daten im Request größer werden und bereits übertragene Daten erneut übertragen werden müssen, da immer alle relevanten Daten übertragen werden müssen. Auf die Daten vorheriger Requests kann nicht zugegriffen werden. Die Zustandslosigkeit bringt die Eigenschaften Sichtbarkeit, Ausfallsicherheit und Skalierbarkeit mit sich. Die Sichtbarkeit ergibt sich daraus, dass ein einzelner Request alle relevanten Daten enthält und keine Annahmen über vorherige Requests getroffen werden müssen.

Die Ausfallsicherheit wird verbessert, da bei einem partiellen Ausfall des Systems nur der letzte Request wiederholt werden muss. Die Skalierbarkeit wird verbessert, weil keine Daten auf dem Server gehalten werden und somit kein Session-Sharing benötigt wird.

7.2 JAX-RS mit Jersey

In der JSR 311 wird die JAX-RS (Java API for RESTful Webservices)-Spezifikation definiert. JAX-RS

hat folgende Ziele:

- Die Grundlage der Entwicklung sind POJO, die per Annotation als Webresource zur Verfügung gestellt werden. Die Spezifikation legt auch den Lebenszyklus und den Sichtsbarkeitsbereich (Scope) der Resource fest.
- JAX-RS basiert auf HTTP. Eine Unabhängigkeit des Protokolls wird nicht angestrebt.
- JAX-RS strebt eine Unabhängigkeit des Formats an. Diese Formate werden per MIME-Type im Header angegeben und definieren so den erwarteten Content-Type.
- JAX-RS ist unabhängig von einem Container. Die Spezifikation unterstützt das Deployment in einem Servlet-Container und in einer JAX-WS-Umgebung. JAX-RS verwendet folgende Terminologie:

Resource class	Eine Java-Klasse, die per Annotations eine Webresource implementiert.
Resource method	Eine Methode einer Resourcenklasse, die einen spezifischen Request verarbeitet.
Provider	Die Implementation eines JAX-RS-Interfaces. Referenz-Implementation: Jersey

Die JSR-311 definiert folgende Annotations:

Annotation	Element	Beschreibung
`@Consumes`	Klasse oder Methode	Liste der Mediatypen, die konsumiert werden können. `@Consumes("application/x-www-form-urlencoded")`
`@Produces`	Klasse oder Methode	Liste der Mediatypen, die erzeugt werden können. `@Produces("text/plain")`
`@GET @POST @PUT @DELETE @HEAD`	Methode	Spezifiziert, dass diese Methode einen entsprechenden Request behandelt.
`@Path`	Klasse oder Methode	Spezifiziert den relativen Pfad zu dieser Resource. Angabe aus der Klasse und den Methoden werden zusammengesetzt. `@Path("info")`
`@PathParam`	Parameter, Feld oder Methode	Spezifiziert, dass ein Teil des Pfades als Parameter übergeben wird, z.B. `http://beispiel.org/Lieferanten/l321`. wird l321 als Lieferantennummer bei `getLieferant(@PathParam("nr") String liefNr)`
`@QueryParam`	Parameter, Feld oder Methode	Spezifiziert, dass ein bestimmter Query-Parameter zu einer Variablen gemapped wird, z.B. `http://beispiel.org/Lieferanten/s=Zypern` zu der Methode `getLieferanten(@QueryParam("s") String such)`

`@FormParam`	Parameter, Feld oder Methode	Wie @QueryParam nur für Formular-Elemente. Sollte nur bei Methoden-Parametern genutzt werden. `neu(@FormParam("name") String name, @FormParam("nr") String nr)`

`@MatrixParam`	Parameter, Feld oder Methode	Wie @QueryParam nur für Matrix-Parameter.
		
		`@MatrixParam("info")`
		`@DefaultValue("Life")`
		`@Encoded`
		`private String info;`

`@CookieParam`	Parameter, Feld oder Methode	Spezifiziert, dass ein Methoden-Parameter durch einen Cookie-Parameter gefüllt wird.
`@HeaderParam`	Parameter, Feld oder Methode	Überträgt einen Header-Parameter an einen Methoden-Parameter.
`@Encoded`	Klasse, Konstruktor, Parameter, Feld oder Methode	Die Parameter werden normalerweise decoded. Sollte dies nicht geschehen, kann diese Standardeinstellung mit `@Encoded` ausgeschaltet werden.
`@DefaultValue`	Parameter, Feld oder Methode	In Kombination mit den Annotations `@QueryParam`, `@MatrixParam`, `@CookieParam`, `@FormParam` und `@HeaderParam` kann diese Annotation den Vorgabewert spezifizieren.
`@Context`	Parameter, Feld oder Methode	Definiert ein Ziel für eine Dependency Injection, wie im vorherigen Abschnitt beschrieben wurde. `getUriInfo(@Context UriInfo info)`
`@Provider`	Klasse	Annotation für eine Klasse, die eine JAX-RS-Extension-Schnittstelle implementiert.

Die Rückgabe des Ergebnisses erfolgt meist in einem Response-Objekt. Mit dieser Methode kann man auch die Response Codes (200 für ok, 404 für Not Found etc.) zurück geben. Eine nette Übersicht für die Codes findet ihr unter http://www.webmaster-eye.de/Status-Codes-beim-HTTPResponse.149.artikel.html Das maßgebliche Dokument befindet sich unter http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html .

Nutzung von Jersey im Tomcat

Jersey ist die Default-Implementation von JAX-RS. Der folgende Anwendungsfall beschreibt die notwendigen Schritte zur Nutzung von Jersey (Version 2.28) mit dem Tomcat (Version 8.5) in Eclipse.

Kurzbeschreibung

Dieser Anwendungsfall erstellt eine Webapplikation mit einer einfachen Resource zur Überprüfung der Kommunikation. Die Webapplikation wird im Tomcat deployed.

Ausgangssituation

Tomcat ist installiert.

Ziel

Ein einfacher REST-Service ist deployed und kann per URI getestet werden.

Ablauf

Schritt 1: Installation von Jersey

Unter der Adresse http://repo1.maven.org/maven2/org/glassfish/jersey/bundles/jaxrs-ri/2.28/jaxrs-ri-2.28.zip kann man das komplette Archiv für Jersey herunterladen. Entpacke das Archiv an einen Ort deiner Wahl.

Schritt 2: Erzeugen des Projekts

Erzeuge ein neues Projekt für eine Web Applikation (Dynamic Web Project) in Eclipse. Nutze als ContextPath /rs. Stelle deiner Webapplikation die Bibliotheken aus api, ext und lib zur Verfügung. Kopiere die Bibliotheken dafür in das lib-Verzeichnis unter WEB-INF. Nehme die Bibliotheken in den Build-Path auf (erfolgt automatisch).

Schritt 3: Frontcontroller definieren

Binde in der web.xml den ServletContainer von Jersey (bis Jersey 1.16 `com.sun.jersey.spi.container.servlet.ServletContainer` in der Version 2.28 die Klasse `org.glassfish.jersey.servlet.ServletContainer`) als Servlet ein, sorge dafür, dass er beim Start der Applikation gleich geladen wird (`<load-on-startup> 1</load-on-startup>`) und mappe ein passendes URL-Pattern zu dem Servlet (in meinem Beispiel nehme ich /api/*) .

Schritt 4: Erzeugen der Resource-Class

Erzeugen Sie im Paket rest.basic die Resource-Class KommunikationsRestService und annotieren Sie die Klasse mit der @Path- Annotation (`@Path("/basic")`).

Schritt 5: Erzeugen der Resource-Method

Erzeugen Sie die Resource-Method `public String ping(String text)` und annotieren Sie diese mit `@GET`, `@Produces("text/plain")` und einer `@Path("ping")`- Annotation. Der Parameter kann noch mit `@QueryParam("text")` annotiert werden. Treten Sie den Beweis des Aufrufs an, indem Sie den übergebenen String in Großbuchstaben umgewandelt zurückgeben.

Schritt 6: Deployment und Test

Deployen Sie die Webapplikation auf Ihrem Webserver. Testen Sie den Service im Browser durch Eingabe des URL

http://localhost:8080/<Context>/<URLMapping>/<Resource-Class>/<Resource-Method>?text=Test

Dabei ist `<context>` der Kontext der Webapplikation (rs), `<URL-Mapping>` das Mapping aus der `web.xml` (api) zum ServletContainer-Servlet, `<Resource-Klasse>` der Inhalt der Path- Annotation der Resource-Class (basic) und `<Resource-Method>` der Inhalt der Path- Annotation der Resource-

Method (ping). Mit `?text=Test` kann noch ein Parameter übergeben werden. Der Browser sollte nun das erwartete Ergebnis zeigen: TEST.

In unserem Beispiel also:

`http://localhost:8080/rs/api/basic/ping?text=Test`

Alternativen

Jersey kann auch per Maven installiert werden. Der URL für Jersey und Maven ist `http://download.java.net/maven/2/com/sun/jersey/` . Natürlich kann die Ping-Methode auch andere Ausgaben (Repräsentationen) erstellen. Schauen Sie sich die Alternativen mit

- @Produces("text/html")
- @Produces("application/json")
- @Produces("application/xml")

an.

Der Service kann auch per `curl` getestet werden, was den Vorteil hat, dass man den Medientyp besser angeben kann:

```
curl -i http://localhost:8080/rs/api/basic/ping?text=Test –H "ACCEPT:text/plain"
```

Tipp: Erzeugen Sie für die Rückgabe ein Objekt der Klasse `javax.ws.rs.core.Response` und nutze für die Medientypen die Klasse `javax.ws.rs.core.MediaType`.

```
@GET
@Path("ping")
@Produces( MediaType.TEXT_PLAIN)
public Response pingPost(@QueryParam("text") String text){
    return Response.ok(text.toUpperCase(), MediaType.TEXT_PLAIN).build();
}
```

Dokumente

Im Deployment Descriptor web.xml wird der ServletContainer eingebunden und mit einem Mapping verbunden:

```
<servlet>
    <display-name>JAX-RS REST Servlet</display-name>
    <servlet-name>REST-Servlet</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
        <param-name>jersey.config.server.provider.packages</param-name>
        <param-value>de.kiltz.rest.basic</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>REST-Servlet</servlet-name>
    <url-pattern>/api/*</url-pattern>
</servlet-mapping>
```

Die Resource-Class enthält hier nur die eine Resource-Method mit den entsprechenden Annotations:

```
package rest.basic;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
@Path("basic") // 1
public class KommunikationsRestService {
@GET // 2
@Path("ping") //3
@Produces("text/plain") 4
    public String ping(@QueryParam("text") //5
                              String text) {
         return text.toUpperCase();
    }
}
```

Die Resource-Class ist ein normales POJO. Die Path-Annotation bei der Klasse (1) ist das Präfix für jeden weiteren Pfad, der bei den Methoden definiert ist. Die GET-Annotation (2) legt die HTTPMethode fest. Mit der Path-Annotation (3) wird der Pfad der Klasse erweitert. Die Produces-Annotation (4) bestimmt den Rückgabetyp, der vom Empfänger per Accept erlaubt sein muss. Mit QueryParam kann man die Namen der Query-Parameter mit den Parametern der Methode verknüpfen (5).

Ein Gradle Build-Script für diese Konstellation könnte folgendermaßen ausschauen:

```
apply plugin: 'java'
apply plugin: 'idea'
apply plugin: 'eclipse'
apply plugin: 'war'
sourceCompatibility = 1.8
repositories {
    mavenCentral()
}
dependencies {
    providedCompile group: 'javax.servlet', name: 'servlet-api', version: '2.4'
    providedCompile group: 'javax.servlet', name: 'jsp-api', version: '2.0'
    compile fileTree(dir: 'libs/jaxrs-ri/api', include: ['*.jar'])
    compile fileTree(dir: 'libs/jaxrs-ri/lib', include: ['*.jar'])
    compile fileTree(dir: 'libs/jaxrs-ri/ext', include: ['*.jar'])
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
task deploy (type:Copy, dependsOn: build) {
    from('build/libs'){
        rename 'jax-rs-server.war', 'rs.war'
    }
    into "${project.property('tomcat.home')}/webapps/"
    println "kopiere rs.war nach ${project.property('tomcat.home')}/webapps/"
}
```

Dazu könnte man in einer `gradle.properties` noch `tomcat.home` spezifizieren.

Application und @ApplicationPath

Alternativ zum Einbinden in der web.xml oder im JEE-Umfeld kann die Klasse `javax.ws.rs.core.Application` genutzt werden.

```
package rest.basic;
import javax.ws.rs.core.Application;
import javax.ws.rs.ApplicationPath;
@ApplicationPath("/api")
public class RestApplication extends Application {
    @Override
    public Set<Object> getSingletons() {
        HashSet<Object> set = new HashSet();
        set.add(new KommunikationsRestService());
        return set;
    }
}
```

7.3 Request, Response

Dieses Kapitel befasst sich mit den Mechanismen, mit denen wir Informationen aus dem HTTPRequest in Java verarbeiten können und wie wir aus Java heraus unseren Response gestalten können.

Routing zu einer Java-Methode

Um einen spezifischen Request von einer bestimmten Java-Methode behandeln zu können benötigen wir hier ein Routing, das hauptsächlich durch drei Komponenten vorgenommen wird:

- Die Path-Annotation
- Die Request-Methode
- Der Content-Type

Die Path-Annotation

Wie wir in dem Jersey-Kapitel schon kennen gelernt haben, setzt sich unsere Request-URI aus mehreren Teilen zusammen. Nach den Information für Server, Port, Context und URI-Mapping spezifizieren wir mit dem Path die Resource Class und evtl. weiter die Methode.

Normalerweise ist der Value der Path-Annotation ein einfacher String

```
@Path("/kunden)
```

Der durch Path-Variablen erweitert werden kann

```
@Path("images/{image}")
```

Auf diese Variable kann dann mit der PathParam-Annotation (s.u.) zugegriffen werden.

Path-Variablen können auch mit Requlären Ausdrücken erweitert werden:

```
@Path("/kunden")
public class KundenRestService {
    @GET
    @Path("{id : \\d+}")
    public Kunde getKundePerId(@PathParam("id") int id) {
        //...
    }
}
```

Diese ID darf nur numerische Zeichen enthalten. Es kann vorkommen, dass wir mehrdeutige Path-Ausdrücke definieren:

```
@Path("/kunden")
public class KundenRestService {
    @GET
    @Path("{id : .+}")
    public Kunde getKundePerId(@PathParam("id") String id) {
        //...
    }
    @GET
    @Path("{id : .+}/adressen")
    public List<Adresse> getAddressenEinesKunden(@PathParam("id") String id) {
        //...
    }
}
```

Der Aufruf `/kunden/Hägar/adressen` würde sowohl zu der ersten Definition als auch zur zweiten Definition passen. Hier entscheidet Jersey sich für den Ausdruck, der die meisten fixten Treffer hat, hier also für die Methode `getAddressenEinesKunden`.

Die Request-Methode

Die Request-Methoden haben wir im ersten Kapitel schon kennen gelernt:

GET	GET-Aufrufe sind nur lesend und idempotent
POST	Erzeugung einer neuen Resource, nicht idempotent.
PUT	Änderung einer Resource, idempotent.
DELETE	Löschen einer Resource, idempotent.

Sofern eine Request-Methode nicht definiert ist, wird vom Server der Fehlercode Method not allowed (405) zurück gegeben. Ausgenommen hierfür ist es, wenn zu einer Path-Definition gar keine HTTP-Methoden defniert sind. Dann handelt es sich um einen Subresource Locator, der die Resource-Klasse zurück gibt, die eine weitere Verarbeitung des Requests vornehmen soll. Siehe dazu z.B. https://dennis-xlc.gitbooks.io/restful-java-with-jax-rs-2-0-2rd-edition/content/en/part1/chapter4/subresource_locators.html

Der Content-Type

Weiterhin ist für die Auswahl der richtigen Methode die Inhalte der Accept und Content-Type Header bedeutend.

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public RootDatenTypen getDatenPerJSON() {
    // ...
}
@GET
@Produces(MediaType.APPLICATION_XML)
public RootDatenTypen getDatenPerXML() {
    // ...
}
```

Bei einem Accept: `application/json` würde nun die Methode `getDatenPerJSON` und bei Accept:

`application/xml` die Methode `getDatenPerXML` gewählt. Ein Accept: `text/plain` würde automatisch zu einem 406 "Not Acceptable" führen.

JAX-RS Injection

Um Informationen aus dem Request in ein Java-Objekt zu übertragen bietet JAX-RS mehrere Möglichkeiten an:

@PathParam	Ein Teil des Pfades wird als Parameter betrachtet.	`@Path("{id}") … getPerId(@PathParam("id") int id)`

@MatrixParam	Matrix-Parameter sind den QueryParam ähnlich, werden aber mit einem Semikolon getrennt und gehören zu einem Path-Segment, sie sind mit Attributen vergleichbar	`/jacken;farbe=schwarz`
		
		`@MatrixParam("farbe")`

@QueryParam	QueryParam übertragen einzelne Parameter, die als Query-String bei der URL mit übergeben werden.	`GET /dinge?start=0&size=10`
		
		`get(@QueryParam("start") int start, @QueryParam("size") int size)`

@FormParam	werden mit dem Access-Header application/xwww-form-urlencoded aus HTML-Formularen gepostet	analog zu QueryParam, nur aus HTML-Form heraus übertragen.
@HeaderParam	Zugriff auf einen spezifischen Header-Parameter	`get(@HeaderParam("Referer") String referer)`
@CookieParam	Zugriff auf einen Cookie-Parameter der mit NewCookie gesetzt wurde.	`@CookieParam("id") int id)` oder auch `@CookieParam("id") Cookie id`
@BeanParam	Zur Verschlankung der Resource-Method-Signatur kann ein BeanParam angegeben werden, der die Request-, Form- und Header-Paramter kapselt.	
@Context	Mit der Context-Annotation kann man z.B. auf folgende Informationen zugreifen.	`ServletContext, HTTPRequest, UriInfo, ResourceInfo, HTttpHeaders, Providers`

Die Annotations (außer Context) können noch mit den beiden Annotation DefaultValue und Encoded kombiniert werden.

Damit eine automatische Umwandlung der Request-Informationen in die entsprechenden Parameter erfolgen kann ist eine der folgende Voraussetzungen zu erfüllen:

1. Der Parameter ist ein primitiver Datentyp
2. Die Java-Klasse hat einen "Ein-String-Konstruktor"
3. Die Klasse hat eine Methode static <T> valueOf(String s) (z.B. Enums)
4. Das Ziel ist eine List<T>, Set<T> oder SortedSet<T> wobei <T> Die Kriterien aus 2 oder 3 erfüllt.

Content Handler

Um den Body einer Request-Message zu lesen oder einen Body eines Responses zu erzeugen benötigen wir einen JARX-RS Content Handler.

JAX-RS bringt dazu folgende Content Handler mit:

- StreamingOutput
- InputStream, Reader
- File (Input und Output)
- MultivaluedMap (Form, Input und Output)

Hierzu gibt es genügend Beispiele im Netz wobei ich auf die MultivaluedMap besonders hinweisen möchte, da diese wieder eine elegante Möglichkeit zur Stabilisierung der Schnittstelle bei `@FormParam` zur Verfügung stellt.

Gebräuchlicher ist die Nutzung von JAXB zur Umwandlung von Java-Datenstrukturen zu einem Massage Body. Hierzu muss man sein Projekt durch einen JAXB-Handler erweitern. Eine gute Wahl hierbei ist der Jackson JAXB Provider. JAXB kann sowohl XML- als auch JSON-Formate erstellen. In den meisten Fällen ist das JSON-Format eine gute Wahl, da es performanter und kleiner ist.

Mit dem MessageBodyReader und dem MessageBodyWriter kann man auch seinen eigenen ContentProvider erzeugen.

Responses

Daten

Mit der Nutzung von JAX-RS i.V. mit JAXB können die Methoden unsere Datenobjekte direkt zurück geben:

```
@GET
@Produces({ MediaType.APPLICATION_JSON })
List<Kunde> getKunden(@QueryParam("s") String suchBegriff);

@GET
@Produces({ MediaType.APPLICATION_JSON })
@Path("{id}")
Kunde getKunde(@PathParam("id") long id);
```

Dabei ist die Klasse Kunde eine JAXB-Klasse, die sowohl in JSON als auch in XML zurück gegeben werden kann.

Alternativ kann man die Rückgabe auch in Response-Objekt verpacken:

```
@Override
public Response getKunde(long id) {
    Kunde k = service.getKunde(id);
    return Response.ok().entity(k).build();
}
```

Bei komplexeren Entitys kann man diese auch in der Klasse `GenericEntity` verpacken.

```
//...
    List<Kunde> liste = ...
    GenericEntity entity = new GenericEntity<List<Kunde>>(liste) {};
    return Response.ok().entity(entity).build();
```

Fehler und Exceptions

Jede Resource-Methode darf eine checked oder unchecked Exption werfen. diese wird dann automatisch in einen HTTP-Error 500 umgewandelt.

Um den Fehler genauer zu kontrollieren bietet sich die Klasse WebApplicationException, der man den entsprechenden Fehlerstatus mit übergeben kann:

```
@GET
@Path("images/{image}")
@Produces("image/*")
public Response getImage(@PathParam("image") String image,
            @Context ServletContext ctx) {
    File f = new File(ctx.getRealPath("/img/")+image);
    System.out.println(f.getAbsolutePath());
    if (!f.exists()) {
        throw new WebApplicationException(404);
    }
    String mt = new MimetypesFileTypeMap().getContentType(f);
    return Response.ok(f, mt).build();
}
```

`WebApplicationException` hat interessante Unterklassen, die die typischen Fehlersituationen abbilden. Statt der `new WebApplicationException(404);` hätte man auch eine `new NotFoundException();` werfen können.

Die Hierarchie von WebApplicationException:

```
Exception
-> RuntimeException
    -> WebApplicationException
        -> RedirectionException
        -> ServerErrorException
            -> ServiceUnavailableException
            -> InternalServerErrorException
        -> ClientErrorException
            -> NotAcceptableException
            -> ForbiddenException
            -> NotAuthorizedException
            -> BadRequestException
            -> NotAllowedException
            -> NotFoundException
            -> NotSupportedException
```

7.4 Client-API

JAX-RS Client

JAX-RS bietet eine Klasse Client, die mit Hilfe eines ClientBuilder erstellt wird. Der Client kann wieder verwendet werden, muss aber kontrolliert am Ende geschlossen werden.

Bei der Erzeugung des Clients kann man auch gleich zusätzliche Provider wie z.B. den JacksonJsonProvider registrieren.

Aus dem Client kann man dann mit einer URI und möglichen Parametern ein WebTarget erstellen, auf das man dann den eigentlichen Request ausführen kann:

```
public class BasicTest {
    private static final String URL = "http://localhost:8081/rs/api/basic/";
    private static Client client;

    @BeforeClass
    public static void init() {
        client = ClientBuilder.newClient().register(new JacksonJsonProvider());
    }

    @AfterClass
    public static void beende() {
        client.close();
    }

    @Test
    public void testPing() {
        String query = "Test";
        String matrix = "JUnit";
        WebTarget target = client.target(URL).path("ping").matrixParam("info",
            matrix).queryParam("s", query);
        String resp = target.request().accept(
            MediaType.TEXT_PLAIN).get(String.class);
        Assert.assertEquals(query.toUpperCase()+" "+matrix, resp);
    }
}
```

Weitere REST-Clients

Ein REST-Client kann eigentlich jeder sein, der einen HTTP-Request absenden und auswerten kann.

z. B.:

- eine java.net.URL-Connection
- ein Apache HttpClient
- ein RESTEasy Client Proxy
- AJAX in den verschiedensten Varianten

Zum Testen eignet sich z. B. curl, Insomnia oder Postman (Links s. im Anhang).

7.5 REST API Design

URLs

- URLs sollten im spinal-case definiert werden (alles in Kleinbuchstaben mit einem Bindestrich dazwischen)
  snake_case und CamelCase sind natürlich auch möglich, sollten aber konsequent genutzt werden.
  spinal-case wird von der RFC3986 (URI: Generic Syntax), snake_case wird von den WebGiganten (Google, Facebook, Twitter) bevorzugt.
- URLs sollten im Sinne eines Pfades zu einer Resource gestaltet werden.
  Es sollten keine Verben, sondern Nomen verwendet werden Falls Verben zum Signalisieren von Aktionen doch notwendig sind, sollten sie ans Ende der URL.
  Beispiel: /datensicherung/execute

Resource	GET Read	**POST** Create	**PUT** Update	**DELETE**
/kunden	gibt eine Liste von Kunden zurück	Erzeugt einen neuen Kunden	Bulk update für Kunden	Löscht alle Kunden
/kunden/008	gibt einen bestimmten Kunden zurück	Method not allowed (405)	Update eines speziellen Kunden	löscht einen speziellen Kunden

- GET-Methoden sollten keine Änderungen im Datenbestand vornehmen. Hierfür sind die PUT, POST und DELETE-Methoden zuständig.
- GET, PUT und DELETE sind idempotent, POST ist nicht idempotent.
- Für partielle Updates kann die Http-Methode PATCH genutzt werden.
- Mische keine Singular- und Plural-Nomen, der Einfachheit halber nutzt man durchgängig den Plural.
- Relationen werden durch Sub-Resourcen ausgedrückt:
  GET /kunden/008/umsaetze/2019

- spezifiziere im Header den Content-Type. Content-Type definiert das Request-Format, Accept definiert das Rückgabeformat. Sofern nichts dagegen spricht kann man durchgängig mit `MediaType.APPLICATION_JSON` arbeiten.
- Benutze eine Versionierung - von Anfang an.
  http://beispiel.org/api/v1/kunden

Rückgabe mit HTTP-Responsecodes

In REST sollte man die komplette Vielfalt der HTTP-Responsecodes nutzen. Die wichtigsten Codes sind:

Responsecode	**Wann zurückgeben?**	**Weitere Details**
200 OK	Ausführung der Aktion war erfolgreich.	

201
Created
	Ausführung der Aktion war erfolgreich.	Wird ausschließlich bei den POST Requests nach dem Anlegen eines neuen Datensatzes zurückgegeben. Hierbei ist es üblich die URI und die ID des neue angelegten Datensatzes im zusätzlichen Location-Header zurückzugeben.
		`CURL -X POST \`
		`-H "Accept: application/json" \`
		`-H "Content-Type: application/json" \`
		`-d '{"state":"running","id_client":"007"}' \`
		`https://api.fakecompany.com/v1/clients/007/orders`
		`< 201 Created`
		`< Location:`
		`https://api.fakecompany.com/orders/1234`

204 No Content	Ausführung der Aktion war erfolgreich.	Es wird jedoch kein / einen leeren Response zurückgegeben (entspricht einer void-Methode in Java)

500
Internal Server Error
	Allgemeiner Serverfehler.	Fallback-Fehlermeldung, wenn nichts anderes definiert ist. Fehler-Payloads sollten im speziellen JSON-Format zurückgegeben werden. Beispiel
		`{`
		`"statusCode": "INTERNAL_SERVER_ERROR",`
		`"applicationCode": "32",`
		`"message": "something goes wrong…",`
		`"stackTrace": "java.lang.RuntimeException: das ist`
		`die ursache…"`
		`}`

503 Service Unavailable	Fehler, wenn der Service einen weiteren Knoten nicht erreichen konnte.	
400 Bad Request	Allgemeiner Clientfehler.	Faustregel: wenn der Client einen Fehler gemacht hat und seinen Request umformulieren muss damit dieser erfolgreich ist
401 Unauthorized	Fehler, wenn der Client nicht	

	authentifiziert ist.	
403 Forbidden	Fehler, wenn der Client zwar authentifiziert ist, aber ihm fehlen die Rechte	

Siehe auch https://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/

Jersey stellt uns dafür die Enum `javax.ws.rs.core.Response.Status` zur Verfügung.

Konventionen zum Benennen von Java-Klassen

Man sollte auf eine einheitliche Benennung der Java-Klassen für REST achten. Welche Konventionen nun genau für Ihr Projekt oder Ihr Unternehmen genutzt werden steht Ihnen frei.

Ein Vorschlag dazu schaut so aus:

Benennung der Services:

- <Resource>RestService - für REST Services
- <Resource>SoapService - für SOAP Services

Es bietet sich an gegen ein Interface zu programmieren und im Interface die JAX-RS Annotations zu definieren.

Java-Klassen für JSON-Requests / POJOs

- Sollten im selben Package wie der Service abgelegt sein
- Keine pauschale Präfixe oder Suffixe verwenden
- Sinnvolle Suffixe je nach Anfrage vergeben (VO, DTO, Wrapper, Impl sind NICHT sinnvoll)
- Beispiele:
  KundenInfo bei LeseAnfrage
  KundenInfoMitHistorie für Details zum Lesen etc.

Request-Klassen sind nur für die Web Service-Schicht gedacht und dürfen im Domänen-Modell nicht verwendet werden Ebenso darf in den Signaturen der Rest-Service-Methoden keine Entität aus dem Domänen-Modell auftauchen

8 Architektur

8.1 Die 5 Ebenen in J2EE

Als Ebene (engl. tier) ist hier die logische Zusammenfassung von Komponenten in einem System zu verstehen.

J2EE unterscheidet 5 Ebenen:

Client

Diese Ebene enthält alle Komponenten, die Clients einer Enterprise-Applikation sind, z. B.:

- Web-Browser
- Hand-held devices
- andere Applikationen, die auf Services der Web Applikation zugreifen

Presentation

Diese Ebene enthält die Präsentations-Logik und arbeitet mit der Client-Ebene als Schnittstelle zusammen.

Diese Ebene

- akzeptiert Requests
- behandelt Autorisierung (Genehmigung) und Authentifizierung (Bestätigung)
- managed Sessions
- delegiert den Business-Prozess zur Business-Ebene
- präsentiert die dem Client die geforderte Response

Folgende Komponenten werden hierfür genutzt:

- Filter
- Servlets
- JavaBeans
- JSP
- Utility-Klassen

Business

Diese Ebene ist das Herz der Applikation und implementiert die Kern-Business-Services.

Diese Ebene besteht normalerweise aus Enterprise-JavaBeans-Komponenten, die die Regeln der Business-Verarbeitung handhaben.

Integration

Der Job dieser Ebene ist die nahtlose Integration der unterschiedlichen externen Ressourcen aus der Ressource-Ebene mit der Business-Ebene.

Die Komponenten nutzen u. a. folgende Mechanismen:

- JDBC
- J2EE connector technologie
- propritäre Middelware

Ressource

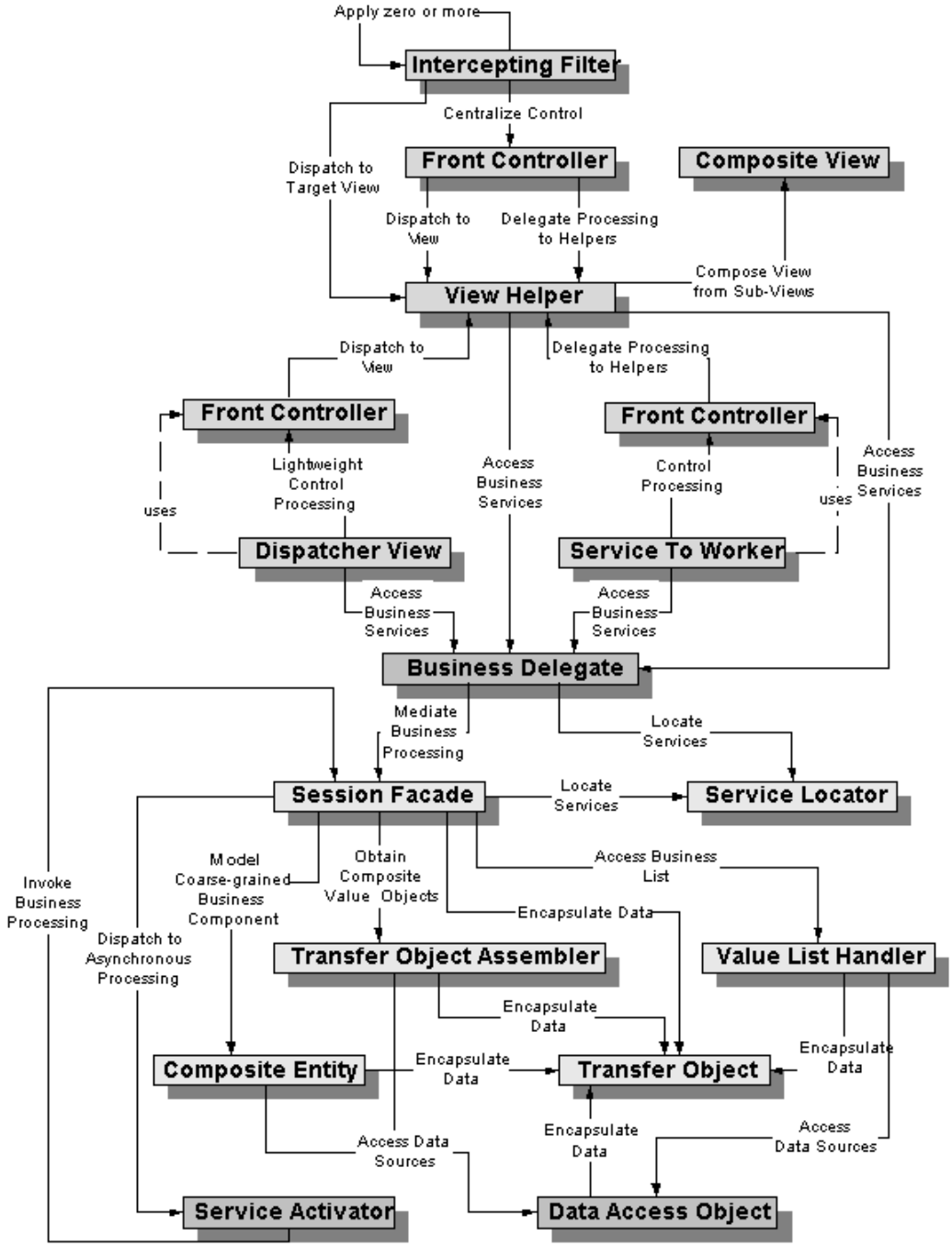
Diese Ebene enthält die externen Ressourcen, die die Daten für die Applikation zur Verfügung halten.

Diese Ressourcen können

- entweder Datenbanken
- oder weitere Systeme wie
    - Applikation auf Mainframes
    - vorhandene Systeme (Altsysteme)
    - B2B-Lösungen
    - Drittanbieter-Services (z. B. Kreditkarten-Autorisierung)

8.2 J2EE Core Patterns

Ein Überblick über die Core J2EE Patterns aus dem Buch Core J2EE Patterns: Best Practices and Design Strategies.



Hierbei ist der BusinessDelegate der Trenner zwischen der Präsentations- und der Business-Ebene.

Das DAO und der Service Activator gehören zu der Integrationsebene.

8.3 MVC - Model-View-Controller

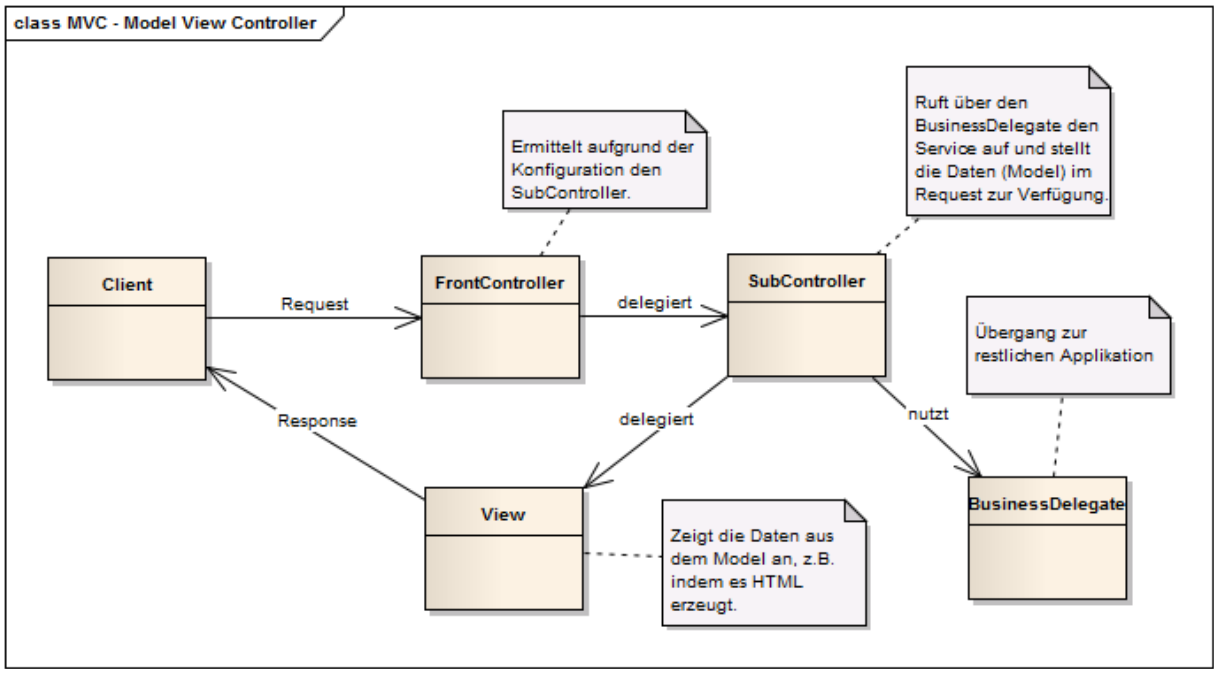
Der Model-View-Controller besteht aus den Teilen (wie der Name schon sagt ;-))

- **Model**: Die Daten, die z.B. durch Service-Methoden ermittelt werden.
- **View**: Die Anzeige und Aufbereitung der Daten.
- **Controller**: Er analysiert den Request kommuniziert mit dem Service und stellt die dadurch gewonnen Daten (das Model) im Request-Scope (oder auch Session-Scope) zur Verfügung.

Vorteil:

Einfache Verwaltung des Programmes durch die Trennung von Verantwortlichkeiten. Der Controller repräsentiert einen Einstiegspunkt in die Applikation, der zentral das Security-Management und das Zustands-Management vornehmen kann.

Somit haben die Designer lediglich die Aufgabe die Daten zu präsentieren, da die JSP keine komplexe Business-Logik beinhalten.



In der Praxis ist dem Controller meist ein Frontcontroller vorgeschaltet, der auf Grund einer Konfiguration (meist XML-Datei) den gewünschten SubController ausfindig machen kann und den Request an ihn weiter leitet.

9 Links & Quellen

9.1 JavaFX

- https://gluonhq.com/products/javafx/ Download JavaFx

9.2 REST

- https://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/
- http://blog.octo.com/en/design-a-rest-api/
- https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api
- https://eclipse-ee4j.github.io/jersey.github.io/documentation/latest/ Jersey User Guide
- https://curl.haxx.se/ curl
- https://insomnia.rest Insomnia
- https://www.getpostman.com Postman

9.3 Bücher

- Anton Epple: JavaFX 8
  ISBN: 978-3-86490-169-0
  Verlag: dpunkt.verlag

- Bill Burke: RESTful Java with JAX-RS 2.0
  ISBN-10: 144936134X
  Verlag: O'Reilly Media