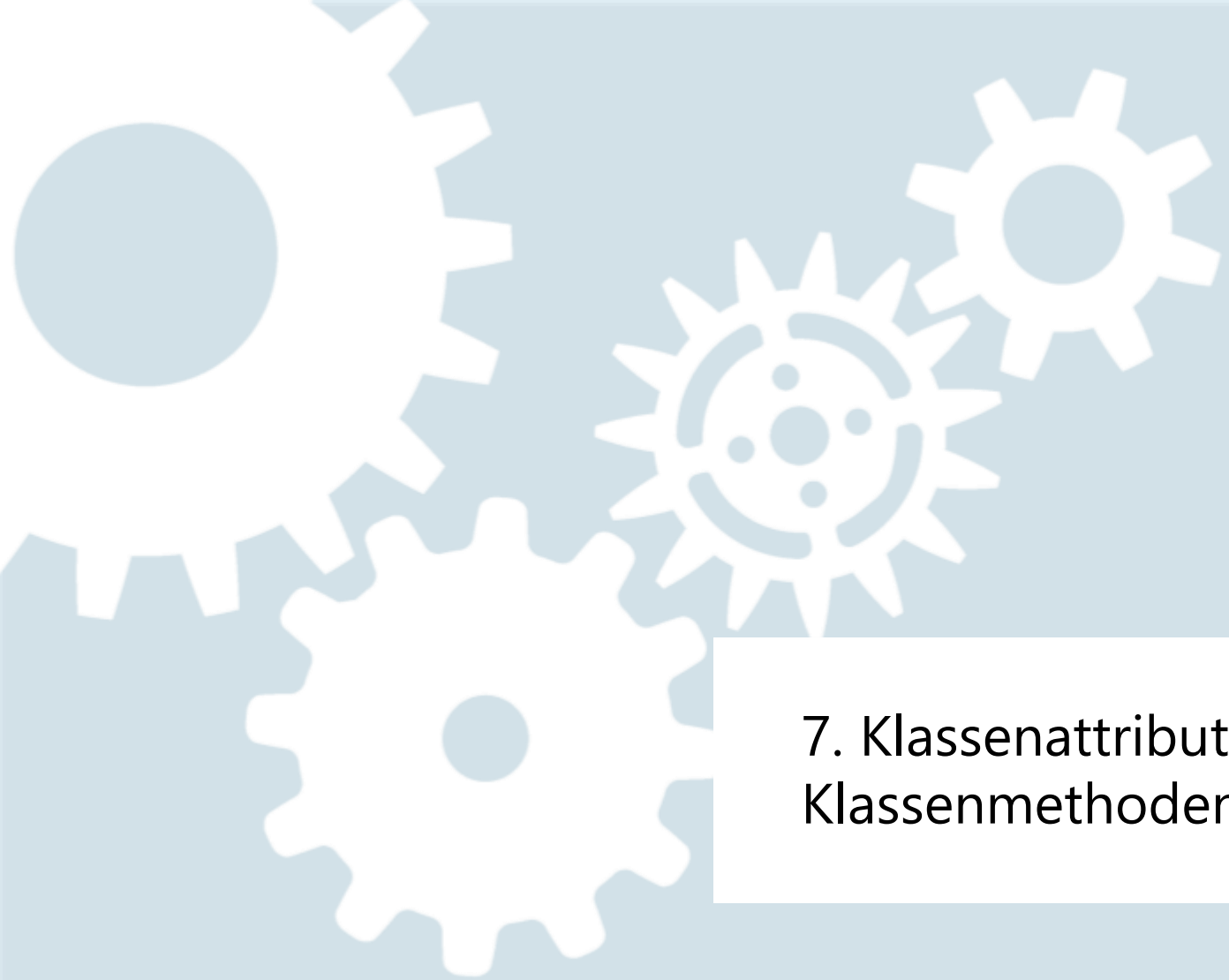


7. Klassenattribute und Klassenmethoden

ARINKO

Attribute (engl. „attributes“ oder gerne auch „fields“):

- Informationen, Daten eines Objekts
- können unveränderlich (konstant) oder variabel sein

Methoden (engl. „methods“):

- Operationen eines Objekts
- können den Objektzustand verändern oder in Abhängigkeit von diesem agieren
- können Nachrichten/Daten (= Parameter) von ihrem Aufrufer empfangen
- können Antworten (= Rückgabewerte) an diesen zurückliefern

- Attribute und Methoden sind zunächst rein objektbezogenen Begriffe.
- In einer Klasse können aber auch Attribute und Methoden definiert werden, die sich nicht auf Objekte sondern auf die Klasse beziehen.
- Dieses Phänomen nennt sich

  Klassen-Attribute und **Klassen-Methoden**

- Während es von Objektattributen mehrere Sätze (je Objekt) gibt, existieren Klassenattribute nur einmal, sie "hängen direkt an der Klasse".
- Während Objektmethoden auf dem "aktuellen Objekt" arbeiten, arbeiten Klassenmethoden auf der Klasse ohne Kenntnis eines spezifischen Objektes.

- Elemente, die sich nur auf die Klasse beziehen sollen, bekommen in Java den Modifizierer

  `static`

- Es gibt verschiedene Herangehensweisen in div. objektorientierten Sprachen, beispielsweise sind bei vielen Smalltalk-Versionen Klassenelemente separat von Instanzelementen.
- In Java kommt alles in der selben Klasse direkt nebeneinander vor.
- Wenn man sich genau ausdrücken möchte, kann man von

  Instanz-Attribut
  Instanz-Methode

  gegenüber

  Klassen-Attribut
  Klassen-Methode

  sprechen.

- Dieses Beispiel zeigt Klassenelemente zur Verwaltung zentraler Daten

Klassenattribut

Klassenmethode

```
public class Konto
{
    private String kontonummer;
    private static double zinssatz = 0.01;
    private int    kontostand;

    public static double getZinssatz()
    {
        return zinssatz;
    }

    public static void setZinssatz(double z)
    {
        zinssatz = z;
    }
…
}
```

- Dieses Beispiel zeigt eine typische Hilfsklasse, von der sicher keine Objekte gebraucht werden

typischerweise haben solche Klassen wenn, dann nur final Attribute

```
public class Berechnungshilfen
{
    public static int runde(double wert)
    {
        double zehntel = 10.00 * wert;
        int zehntelInt = (int) zehntel;
        int letzteZiffer = zehntelInt % 10;
        if (letzteZiffer >= 5)
        {
            zehntelInt = zehntelInt + 10;
        }
        return zehntelInt / 10;
    }
}
```

Klassenmethode als, lax gesprochen, "Durchlauferhitzer", d.h. Wert rein, Wert raus

- Innerhalb von Klassenmethoden kann auf eigene Klassenelemente nur durch Nennung des Namens zugegriffen werden:

```
public static double getZinssatz()
{
    return zinssatz;
}
```

- Innerhalb von Instanz-Methoden kann ebenfalls auf alle eigenen Klassenelemente nur durch Nennung des Namens zugegriffen werden.
- Von außerhalb greift man auf Klassenelemente mit dem Namen der anderen Klasse und dem Zugriffsoperator . zu.

```
public void verarbeiteZinsen()
{
    double neuerStand = this.kontostand * (1 + zinssatz);
    this.setKontostand(Berechnungshilfen.runde(neuerStand));
}
```

Zugriff auf eigenes Element

Zugriff auf externes Element

- In Klassenmethoden hat man Zugriff auf sämtliche Attribute und Methoden einer Instanz der selben Klasse, vorausgesetzt man besitzt eine Referenz auf das Objekt.

  Sonst weiß man ja nicht, welches Objekt gemeint ist!
- Klassenmethoden dürfen, da sie in der Klasse angesiedelt sind, auch sämtliche privaten Elemente des Objekts verwenden.

- Klassenelemente besitzen ebenfalls Zugriffsmodifizierer, d.h. sie können selbst private und public sein.

```
private static double zinssatz = 0.01;

public static double getZinssatz()
{
    return zinssatz;
}
```

- Klassenattribute können ebenfalls mit dem Modifizierer **`final`** versehen werden.
- Typischerweise besitzen diese Attribute einen Initialisierer.

```
public class Konstanten
{
    public static final double AVOGADRO = 6.02214076E23;
}
```

- Sehr häufig kommt die Kombination mit **`public static final`** vor, d.h. es handelt sich um öffentlich sichtbare Klassenkonstanten.

- Haben **`final`** Klassenattribute keinen Initialisierer, dann müssen sie in einem statischen Initialisierungsblock initialisiert werden:

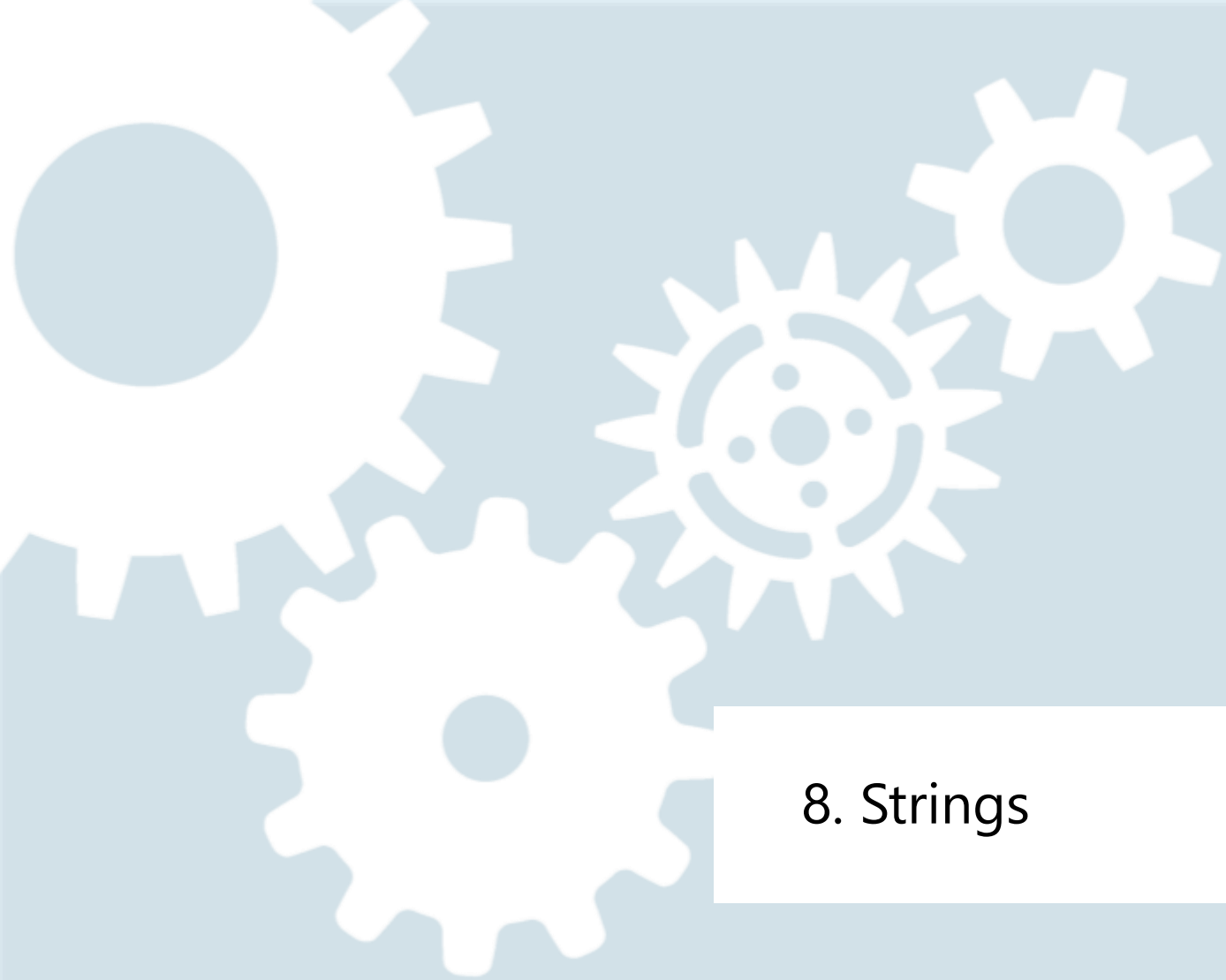
```
public class Konstanten
{
    public static final double AVOGADRO;

    static
    {
        // berechne die AVOGADRO-Konstante oder lese sie
        // aus einem externen Medium ein
        AVOGADRO = readAvogadroValue();
    }
…
}
```

- Der statische Initialisierungsblock erscheint also wie die anderen Initialisierungsblöcke im Klassenrumpf, trägt aber den Modifizierer `static`.
- Natürlich darf es auch von diesen Blöcken mehrere geben.
- Sie werden dynamisch, zum Zeitpunkt wenn die Klasse in die virtuelle Maschine geladen wird, von oben nach unten ausgeführt.

- Klassenelemente werden unterstrichen.

Konto
-kontostand : int -zinssatz : double
<<Konstruktor>> +Konto(stand : int) <<Methoden>> +getStand() : int +getZinssatz() : double +setStand(neuerWert : int)



8. Strings

ARINKO

- Strings ist kein elementarer Typ in Java.
- Sondern Strings sind Objekte der Klasse `String`.
- Intern sind Strings `char`-Arrays, was der Benutzer aber nicht merkt, da dieses Detail komplett verborgen ist.
- Strings besitzen als einziger Objekttyp Literale `"Java-Vorlesung"`
- String-Objekte sind immutable (unveränderbar), d.h. man kann einen existierenden String nicht modifizieren.
- Strings besitzen als einzige Klasse einen für sie veränderten Operator, nämlich + für die String-Konkatenation.

- In einigen OO-Sprachen (C++/C#...) wird eine weitere, objektorientierte Technik angeboten, das sogenannte "Operator Overloading".
- Dies erlaubt, Operatoren für eigene Datentypen anzupassen.
- Beispiel in C#:

```
public static Myclass operator *(Myclass a, Myclass b)
{
  …
}

irgendwo dann:

Myclass X = new Myclass();
Myclass Y = new Myclass();
Myclass Z = X * Y;
```

- Die Syntax war in C++ (C# existierte bei der Sprachdefinition von Java noch nicht) so komplex und fehlerträchtig und die Anwendungsgebiete relativ rar (hauptsächlich komplexe Zahlen und Matritzen), dass dieses Feature nicht in Java aufgenommen wurde.

- Weiteres Beispiel: die Bruchzahl-Aufgabe

```
public static Bruchzahl operator +(Bruchzahl b1, Bruchzahl b2)
{
     long gemeinsamerNenner = b1.nenner * b2.nenner;
     long zaehler1 = b1.zaehler * b2.nenner;
     long zaehler2 = b2.zaehler * b1.nenner;
     return new Bruchzahl(zaehler1 + zaehler2, gemeinsamerNenner);
 }

 public static Bruchzahl operator *(Bruchzahl b1, Bruchzahl b2)
 {
     long neuerNenner = b1.nenner * b2.nenner;
     long neuerZaehler = b1.zaehler * b2.zaehler;
     return new Bruchzahl(neuerZaehler, neuerNenner);
 }

 public static Bruchzahl operator -(Bruchzahl b1, Bruchzahl b2)
 {
     return b1 + b2.bildeGegenwert();
 }

 public static Bruchzahl operator /(Bruchzahl b1, Bruchzahl b2)
 {
     return b1 * b2.bildeKehrwert();
 }
```

hier wird auf den bereits definierten Operator * zurückgegriffen

- Lästig wird es dann, wenn man unterschiedliche Typen mischen möchte…

```
public static Bruchzahl operator *(Bruchzahl b, long l)
{
    return b * new Bruchzahl(l, 1);
}

public static Bruchzahl operator *(long l, Bruchzahl b)
{
    return b * l;
}
```

- Das ist nötig, weil das Kommutativgesetz nicht automatisch gilt…

- Erzeugung

```
String s1 = new String(); // leerer String
String s2 = "Java-Vorlesung";
String s3 = new String(s2);

char[] array = { 'J', 'a', 'v', 'a' };
String s4 = new String(array);
```

- Einige Methoden

```
int laenge = s2.length(); // Länge
int platz = s2.indexOf('-'); // Stelle wo - vorkommt
// die meisten Suchmethoden funktionieren mit char oder String
int letztesA = s2.lastIndexOf("a"); // letztes a
```

- Prä- und Postfix-Methoden

```
boolean beginnt = s2.startsWith("Java");
boolean endet = s2.endsWith("ung");
```

- Zerlegung von Strings

```
String java = s2.substring(0, 4);
String vorlesung = s2.substring(5);
```

- `substring` nimmt einen Beginn-Index und einen optionalen nicht mehr enthaltenen End-Index.

- Strings in Großbuchstaben oder in Kleinbuchstaben umwandeln:

```
String jg = java.toUpperCase();
String vk = vorlesung.toLowerCase();

String mitLeer = "    sowas kommt oft aus ner Datenbank    ";
String ohneLeer = mitLeer.trim();
```

- Wo hat hier der Programmierer einen Fehler versteckt ☺ ?

```
String anfang = s2.substring(0,4);
anfang.toUpperCase();
if (anfang.indexOf("av") >= 0)
{
    // do something
}
```

- String hat einige statischen `valueOf()` Methoden für die Umwandlung elementarer Daten in einen String.

```
int alter = 42;
String alterAlsText = String.valueOf(alter);
```

- Für die Konkatenation (Aneinanderhängen) von Strings existiert der + Operator.
- Er erlaubt auch eine indirekte Umwandlung von elementaren Daten.

```
int alter = 42;
String alterAlsText = String.valueOf(alter);

String text1 = "Sie ist " + alterAlsText + " Jahre alt.";
String text2 = "Sie ist " + alter + " Jahre alt.";
```

- Ratespiel: was steht hier in t1 und t2?

```
String t1 = 40 + 2 + " Jahre ist sie alt.";
String t2 = "Sie ist " + 40 + 2 + " Jahre alt";
```

- Jede Klasse darf eine Methode enthalten, die dafür da ist, eine String-Repräsentation ihrer Objekte zu erhalten.
- Sehr gerne wird das für einfache Visualisierungen oder öfters noch für Debug-Zwecke gebraucht.
- Es empfiehlt sich daher, in wichtigen Klassen diese Methode zu implementieren.
- Die Methode muss folgende Signatur haben:

```
public String toString()
```

- Es ist also nicht die Aufgabe der Methode, eine Ausgabe zu machen, sondern lediglich einen String zusammenzubauen.
- Ist keine eigene Methode implementiert, erhält man die Standardimplementierung, die den Klassennamen gefolgt vom Klammeraffen und einem Hashcode liefert: `MyClass@1ab3fdc`
- Wird in einer String-Konkatenation mit `+` eine Referenz eines beliebigen Objektes verwendet, wird die `toString()`-Methode ausgeführt.
- Auch die `print()` und `println()`-Methoden von `System.out` verwenden sie.

- Hier eine menschenlesbare Implementierung:

```
public String toString()
{
  return "Konto " + this.kontonummer + ", Stand: " + this.kontostand + " EUR";
}
```

- IDEs wie Eclipse generieren gerne eine etwas technischere Variante:

```
public String toString()
{
 return "Konto [kontonummer=" + kontonummer + ", kontostand=" + kontostand + "]";
}
```

- Beispiel für die implizite Verwendung:

```
Konto konto1 = new Konto("DE68-0815-4711", 848);
System.out.println(konto1);
String satz = "Es handelt sich um " + konto1;
System.out.println(satz);
```

implizite Verwendung

implizite Verwendung

- Mit Strings sind seit Java 5 für die Low-Level-Formatierung weitere Möglichkeiten verfügbar.
- Dabei hat man sich an das aus C bekannte printf-Konstrukt angelehnt.

```
public class Formatierung
{
    private static Konto[] konten = {
         new Konto("0815", 0.02, 150),
         new Konto("4711", 0.03, 300),
         new Konto("N-EU")
    };

    private static String ausgabetext =
      "Konto '%s' hat einen Stand von %5d EUR und einen Zinssatz von %4.1f%%.%n";

    public static void main(String[] args)
    {
        for (Konto konto : konten)
        {
            System.out.printf(ausgabetext, konto.getKontonummer(),
                              konto.getKontostand(), konto.getZinssatz()*100);
        }
    }
}
```

Format einer Formatierungsschablone

- Allgemeine Form

```
%[argument_index$][flags][witdh][.precision]conversion
```

- Dabei bedeutet:

argument_index:	der Index (+1) im Parameterarray
flags:	Zusatzangaben, die vom conversion-Typ abhängen
width:	Minimale Anzahl auszugebender Zeichen
precision:	Anzahl an Nachkommastellen (für Kommazahlen)
conversion:	Datentypabhängige Darstellung (Zahl, Datum, Text…)

- siehe https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html

b	Boolescher Wert
h	Hashcode im Hexadezimalformat
s	Wenn das Objekt Formattable ist, wird formatTo aufgerufen, sonst toString.
c	Ein Zeichen
d	Integer, dezimal
o	Integer, oktal
x	Integer, hexadezimal
e	Fliesskommazahl, wissenschaftliche Darstellung
f	Fliesskommazahl, "normale" Darstellung
g	Fliesskommazahl, wechselt zwischen f und e bei großen Exponenten
a	Fliesskommazahl, hexadezimal
t	Datums- und Zeitformat
n	Zeilenumbruch
%	%-Zeichen

```
"Konto '%s' hat einen Stand von %5d EUR und einen Zinssatz von %4.1f%%.%n"
```

5 Stellen, Dezimalzahl

insgesamt 4 Stellen (1 für Komma), 1 Nachkommastelle, Fließkommazahl

```
Konto '0815' hat einen Stand von   150 EUR und einen Zinssatz von  2,0%.
Konto '4711' hat einen Stand von   300 EUR und einen Zinssatz von  3,0%.
Konto 'N-EU' hat einen Stand von     0 EUR und einen Zinssatz von  1,0%.
```

```
"%2$09d EUR sind bei einem Zinssatz von %3$5.2f%% auf Konto '%1$8s'.%n"
```

2. Argument, 9 Stellen, führende Nullen, Dezimalzahl

3. Argument, 5 Stellen, 2 Nachkommastellen, Fließkommazahl

```
000000150 EUR sind bei einem Zinssatz von  2,00% auf Konto '    0815'.
000000300 EUR sind bei einem Zinssatz von  3,00% auf Konto '    4711'.
000000000 EUR sind bei einem Zinssatz von  1,00% auf Konto '    N-EU'.
```

- Direkte Bildschirmausgabe mit printf:

```
System.out.printf(schablone, array);
```

- Formatierung eines Strings mit einer Klassenmethode:

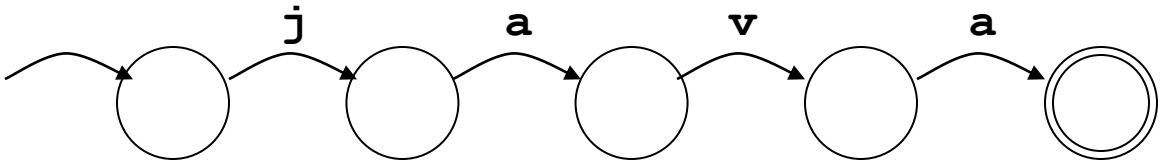
```
String ausgabe = String.format(schablone, array);
```

- Bemerkung: es gibt kein `System.out.printfln();`

- Formatierte Ausgabe (siehe String, String.format) und auch weitergehende Möglichkeiten im Paket java.text sind ein häufiges Thema in Programmen.
- Nahezu ebenso häufig ist die Problematik, Strings zu zerlegen.
- Die Möglichkeiten hierzu mit `substring()` sind begrenzt.
- Der Entwickler greift hier in vielen Programmiersprachen gerne zu "regular expressions".
- In einigen Sprachen (z.B. JavaScript oder Perl) sind Regular Expressions sogar Sprachbestandteil.
- Andere Sprachen, wie Java oder C#, liefern solche Dinge in Form von Paketen/Modulen/Klassen in Namensräumen aus.
- In Java gibt es dafür das Paket **java.util.regex**.

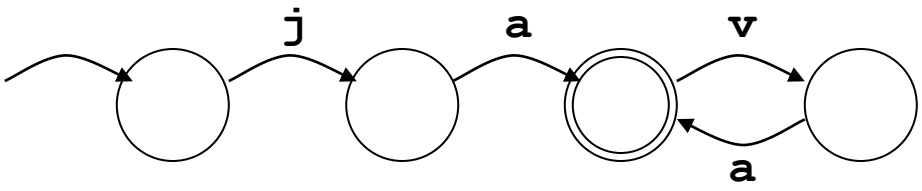
- Sicher schon bekannt, also nur zur Auffrischung…

- Reguläre Ausdrücke entstammen der Automatentheorie und sind beweisbar äquivalent zu (Nicht-)Deterministischen Endlichen Automaten NFA, DFA.



Deterministischer, endlicher Automat, der das Wort **"java"** erkennt.

- Diese Automaten erkennen, kurz gesagt, Wörter bzw. Wortmuster.



Deterministischer, endlicher Automat, der unter anderem das Wort **"java"** erkennt. Aber auch "**ja**", "**javava**", "**javavava**".
Allgemein:
"**ja(va)***"

- Für die Wörter, die ein Automat erkennt, gibt es eine Kurzschreibweise.
- So muss nicht immer der Graph gemalt werden – uff.
- Diese Kurzschreibweise sind die Regulären Ausdrücke.

Regulärer Ausdruck	Entsprechungen
ja(va)*	ja, java, javava, javavavava
ja(va)+	java, javava, javavavava
ja|va	ja, va
j(a|v)*a	ja, jaa, jva, java, jvaa, jaaaaa, jvva
(java)*	*<nichts>*, java, javajava, javajavajava

- Die meisten Programmiersprachen, die mit regulären Ausdrücken umgehen können, bieten mächtigere Werkzeuge an, die oft über das theoretische Modell hinausgehen (teilweise in die Klasse der kontextfreien Grammatiken).
- Auch Java hat hier einiges zu bieten.

- Einen Schnelleinstieg gibt es bereits in der Klasse String.

```
string.replaceAll(regex, replacement);
string.replaceFirst(regex, replacement);
boolean yesno = string.matches(regex);
```

- Allgemeiner sind die zentralen Klassen Pattern und Matcher.
- Pattern stellt ein (zur Laufzeit nach Erzeugung berechnetes) Muster dar.
- Matcher dient der Überprüfung auf das Muster und ggf. der Zerlegung des Strings.
- Für die Zerlegung muss man im Muster sog. "capturing groups" angeben.

- Im Beispiel werden Kontodaten aus einer Datei (zumindest könnten sie daher stammen) eingelesen und Objekte gebaut.

```
private static String[] textdaten = {
  // nummer,stand,zinssatz
  "0815,5,1.2", "4711,2017,2.4", "4242,42" };

public static void main(String[] args)
{
  Pattern format = Pattern.compile("(\\d*)[,](\\d*)([,](\\d*[.]\\d*))?");
  for (String text : textdaten)
  {
    Matcher m = format.matcher(text);
    if (m.matches())
    {
      String kontonr = m.group(1);
      String stand = m.group(2);
      String prozent = m.group(4);
      if (prozent == null)
      {
        prozent = "1.0"; // default
      }

      System.out.println(kontonr + ", " + stand + " EUR, " + prozent + "%");
    }
  }
}
```

- Pattern hat eine direkte Methode `matches()` für den Vergleich.

  Frage: in welchen Situationen ist diese günstig, wo nicht?

- In Strings dient das \ Zeichen als Escape-Zeichen. Daher muss der \ als \\ erscheinen.

- Pattern und Matcher wehren sich mit Laufzeitfehlern, wenn man die Formatschablone nicht korrekt angibt. Selbes gilt für String.format und Co.


Praxis
Theorie