

JAVA für Fortgeschrittene

Fortgeschrittene Konzepte

© eden market

Autor: Anton Kölbl

Gliederung

1. *Einleitung*
2. Formate
3. Internationalisierung
4. Logging
5. Preferences
6. Innere Klassen
7. Reguläre Ausdrücke

Einleitung

Hier werden verschiedene Konzepte behandelt, die man zur professionellen Softwareentwicklung anwenden kann und die insbesondere bei größeren Projekten hilfreich sind.

Java ist schon von der syntaktischen Struktur her außerordentlich gut geeignet, auch große Programme übersichtlich aufzubauen.

Einleitung

- Formate: Erleichtern Ein-, Ausgabe von Zahlen, Datum usw
- I18n: Erleichtert paralleles Entwickeln für verschiedene Sprachen
- Logging: Hilft bei Laufzeitproblemen
- Preferences: Erleichtern Anpassung an den einzelnen Benutzer
- Innere Klassen: Erlauben Kapselung von Klassen
- Reguläre Ausdrücke: Hilfe für Patternmatching

Gliederung

1. Einleitung
2. *Formate*
3. Internationalisierung
4. Logging
5. Preferences
6. Innere Klassen
7. Reguläre Ausdrücke

Formate

In den meisten Programmen muß mit Datum, Uhrzeit und/oder Preisen umgegangen werden. Die Werte sollen in einem bestimmten Format gelesen und geschrieben werden.

Das Package `java.text` stellt viele unterstützende Klassen zur Verfügung.

Datum, Uhrzeit: Formatiert durch `DateFormat`

Formate

Durch statische Methoden kann man Defaultformate erhalten:

`DateFormat.getDateInstance()`: Nur Datum

`DateFormat.getTimeInstance()`: Nur Uhrzeit

`DateFormat.getDateTimeInstance()`: Beides

Formate

Diese Formatierer können Date- Objekte lesen und schreiben:

```
String text; Date date; ...
```

```
dateFormat.parse(text);
```

```
dateFormat.format(date);
```

Dabei wird die aktuelle Spracheinstellung aus dem Betriebssystem übernommen; d.h. deutsche Eigenarten sind berücksichtigt.

Formate

Man kann auch andere Spracheinstellungen nehmen, indem man explizit ein anderes Locale wählt:

```
DateFormat.getDateInstance(Locale.GERMAN);
```

In der Klasse Locale in java.util sind die gebräuchlichsten Sprachen bzw. Länder schon definiert.

Formate

Durch die Unterklasse `SimpleDateFormat` kann man auch explizit ein Format konstruieren, indem man einen Patternstring angibt:

```
new SimpleDateFormat(„yyyy-MM-dd“);  
new SimpleDateFormat("dd MMM 'um' HH 'Uhr'")  
// liefert Format: 19. Januar um 21 Uhr
```

Falls Strings wörtlich übernommen werden sollen, müssen Sie durch Apostroph markiert werden.

Formate

Die häufigsten Abkürzungen sind:

- y Jahr
- M Monat
- d Tag innerhalb des Monats
- H Stunde (von 0 bis 23)
- m Minute
- s Sekunde

Formate

Es existieren unterschiedliche Formen, z.T. auch textuelle wie Monatsnamen. Falls der entspr. Patternbuchstabe 4-fach notiert wird, wird die volle Form verwendet (MMMM), bei MMM wird der Monat abgekürzt. Bei MM wird der Monat immer zweistellig angezeigt und dazu mit 0 als Präfix versehen, bei M werden bei Bedarf zwei Stellen ausgegeben. Beim Parsen müssen nie führende Nullen angegeben werden.

Übung 1, Aufgabe 1

Formate

Die behandelten Formate arbeiten mit Date-Objekten zusammen, die für einen Zeitpunkt stehen, der intern als long gespeichert wird (Zahl der Millisekunden seit dem 1.1.1970, „epoch“). Ein Date wird meist durch den Defaultkonstruktor erzeugt und steht dann auf dem aktuellen Zeitwert. Der Konstruktor für andere Zeitwerte ist jedoch als deprecated markiert und sollte nicht mehr verwendet werden.

Formate

Dazu wird stattdessen die Klasse `Calendar` aus `java.util` verwendet, die auch andere Zeitsysteme verarbeiten könnte – im Moment gibt es nur eine Unterklasse `GregorianCalendar` für das westliche Zeitsystem.

Einen passenden `Calendar` erhält man durch `Calendar.getInstance()`. Es gibt mehrere Setter-Methoden, um dieses Objekt mit einem anderen Zeitstempel zu versehen.

Formate

Zum Setzen von Jahr, Monat, Tag:

```
cal.set(2005, 0, 23); // 23. Januar 2005
```

Die Eigenschaft `lenient` sollte dann sofort auf `false` gesetzt werden, sonst werden Daten wie der 32. Januar akzeptiert und entsprechend extrapoliert (= 1. Februar), das ist meist nicht gewünscht:

```
cal.setLenient(false);
```

Formate

Um einzelne Bestandteile auszulesen, gibt es die get- Methode, der man eine Konstante wie Calendar.YEAR übergeben muß:

```
int year = cal.get(Calendar.YEAR);
```

Durch getTime erhält man ein Date- Objekt, z.B. um es formatieren zu lassen.

```
Date d = cal.getTime();
```

Formate

Weitere wichtige Methoden:

- after
- before
- add
- roll

Add und roll lassen Additionen zu, verhalten sich aber unterschiedlich, da roll keinen Übertrag in größeren Feldern durchführt (31. Januar + 1 Tag = ?).

Übung 1, Aufgabe 2

Formate

Auch Zahlen, insbesondere Gleitkommazahlen sollen häufig formatiert werden. Dies wird durch die Klasse `NumberFormat` erreicht.

Ähnlich wie bei `DateFormat` kann man die gebräuchlichsten Arten durch statische Methoden mit einem optionalen `Locale`-Argument erhalten:

```
NumberFormat.getPercentInstance();  
NumberFormat.getCurrencyInstance(Locale.US);  
NumberFormat.getNumberInstance();
```

Formate

Die Unterklasse `DecimalFormat` kann explizit gewünschte Darstellung liefern; hier wird im Konstruktor ein Patternstring übergeben.

Die wichtigsten Symbole:

- 0 Ziffer (wird evt. mit 0 aufgefüllt)
- # Ziffer (keine überflüssigen 0)
- . Dezimal-Trennzeichen (wird lokalisiert)
- , Gruppen-Trennzeichen (wird lokalisiert)

Übung 1, Aufgabe 3

Formate

Manchmal sollen Meldungsstrings aus festen und variablen Teilen zusammengesetzt werden.

Dazu kann man ein `MessageFormat` mit einem Formatstring konstruieren, der bis zu 10 Parameter `{0}`, `{1}`, ... `{9}` enthalten kann.

```
String pattern = „Datum der Konferenz: {0}“;  
MessageFormat mf = new MessageFormat(pattern);
```

Formate

Optional kann noch ein Format für den Parameter übergeben werden:

```
mf.setFormat(0, DateFormat.getDateInstance());
```

Beim Ausführen ist dann der Wert als `Object[]` einzusetzen:

```
Object[] params = {new Date()};  
System.out.println(mf.format(params));
```

Formate

Oft ist ein Parameter je nach Anzahl (0, 1 oder mehr) unterschiedlich auszugeben. Dann kann man für diesen Parameter ein ChoiceFormat konstruieren und dieses mit setFormat im MessageFormat an der richtigen Stelle einsetzen. Dazu braucht man einen double-Array der Grenzwerte, in der Praxis meist (0, 1, 2), sowie einen String-Array mit den Formatstrings, z.B.

 {"kein Auto", "ein Auto", "{0} Autos"}, wobei {0} den entsprechenden Parameter anspricht.

Formate

```
double[] limits = {0, 1, 2};
String[] formatStrings =
    {"kein Auto", "ein Auto", "{0} Autos"};
ChoiceFormat cf = new ChoiceFormat(limits,
    formatStrings);

MessageFormat mf = new MessageFormat(
    "In 60 Minuten sah man {0} auf der Straße.");

mf.setFormat(0, cf);
```

Formate

Diese Ausdrücke werden allerdings recht komplex; nicht immer ist dies eine Erleichterung gegenüber dem manuellen Aufbauen eines Strings mit Hilfe eines StringBuffer oder StringBuilders.

Übung 1, Aufgabe 4

Gliederung

1. Einleitung
2. Formate
3. *Internationalisierung*
4. Logging
5. Preferences
6. Innere Klassen
7. Reguläre Ausdrücke

Internationalisierung

Um ein Programm für mehrere Sprachen gleichzeitig zu entwickeln, müssen die Beschriftungen, Meldungsstrings usw. parallel in mehreren Versionen gepflegt werden.

Zur Erleichterung versucht man, hierzu externe Ressourcendateien anzulegen, die leicht in eine neue Sprache übersetzt werden können.

Internationalisierung

In Java kann man Stringwerte in Properties-Dateien auslagern (es gibt andere Möglichkeiten für Java-Objekte). Prinzipiell wird dann immer ein ResourceBundle benötigt, das in mehreren Sprachversionen vorliegen kann.

Für jede Variante eines ResourceBundle braucht man eine zugehörige Klasse, z.B.
GUIResources_de, GUIResources_en,
GUIResources_fr.

Internationalisierung

Man kann auch eine Basisversion `GUIResources` einrichten, die Defaultwerte enthält, sowie noch speziellere Varianten, die zusätzlich zum Sprachcode noch einen Landescode enthalten, z.B. `de_DE` für Deutschland, `de_AT` für Österreich.

Hier würde man alle gemeinsamen Bezeichnungen in `GUIResources_de` stecken und nur die unterschiedlichen in die Landesvarianten.

Internationalisierung

Jedes ResourceBundle ist im Prinzip eine Lookup-Table ähnlich wie eine HashMap.

In `java.util` stehen schon zwei einsatzfähige Implementierungen zur Verfügung:

- `ListResourceBundle`
- `PropertyResourceBundle`

Internationalisierung

In der Praxis wird `ListResourceBundle` nur selten verwendet, da hier die Werte direkt im Quellcode abgelegt werden. Es hat allerdings den Vorteil, daß es beliebige Java- Objekte unterstützt, und könnte z.B. für Farbeinstellungen ideal sein.

Meist verwendet man dagegen Unterklassen von `PropertyResourceBundle`, wobei die Klasse nur einen Konstruktor benötigt, der die Verbindung zur Properties- Datei herstellt.

Internationalisierung

```
public class GUIResources_de extends
    ResourceBundle {

    public GUIResources_de() throws IOException {
        super(GUIResources_de.class.getResourceAsStream("
            GUIResources_de.properties"));
    }
}
```

Internationalisierung

Hier wird die zugehörige Properties- Datei als Stream referenziert; das ist notwendig, damit die Java- Runtime sie auch in einer jar- Datei finden kann; ein normaler FileInputStream würde da nicht funktionieren.

Die Properties- Datei enthält dann nur noch Zeilen der Art „Key=Value“:

`computeButton=Berechnen`

Internationalisierung

Wenn sowohl Bundles als auch Properties- Dateien vorbereitet sind, kann man die Ressourcen im Anfangsstadium des Programms holen und die gewünschten Keys auslesen, z.B. für die Beschriftung der GUI- Elemente:

```
guiResources = ResourceBundle.getBundle("GUIResources",  
Locale.GERMAN);
```

Hier wird das deutsche Bundle geholt.

```
String s = guiResources.getString("computeButton");
```

Internationalisierung

Wenn man in der Methode `getBundle()` das Locale nicht mit angibt, wird das Default- Locale des Computers verwendet; man erhält also automatisch ein richtig angepasstes Programm, falls die Ressourcen vorhanden sind.

Das Angeben des Locales ist notwendig, wenn man zur Laufzeit die Sprachversion umschalten will, z.B. in Reaktion auf eine Auswahl in einer `ComboBox`.

Übung 2, Aufgabe 1

Gliederung

1. Einleitung
2. Formate
3. Internationalisierung
4. *Logging*
5. Preferences
6. Innere Klassen
7. Reguläre Ausdrücke

Logging

Eine rudimentäre Form von Logging hat sicher jeder schon praktiziert: Das Einfügen von Statements der Form

```
System.out.println(„Wert von i: „ + i);
```

wenn unerwartete Werte auftreten.

Es ist allerdings mühsam, diese Statements wieder auszukommentieren, wenn alles gut geht, bzw. die Statements wieder einzuschalten.

Logging

Eine Alternative wäre das Debuggen, doch hier braucht man deutlich mehr Zeit zum Ausführen des Codes.

Eine andere Möglichkeit ist das Einrichten eines Logging-Mechanismus, wie er seit dem JDK 1.4 im Package `java.util.logging` integriert ist; vorher wurde meist das Open Source Tool Log4J verwendet.

Logging

Die zwei grundlegenden Typen sind hier Logger und Handler. Ein Logger kann Meldungen in verschiedenen Schweregraden erzeugen; damit diese Meldungen auch erscheinen, muß ihm mindestens ein Handler zugeordnet sein, z.B. ein ConsoleHandler.

Bei einfachen Programmen kann man den global definierten Logger `Logger.global` verwenden; sonst kann man auch beliebig viele eigene Logger definieren, z.B. über Packagenamen:

Logging

`Logger.getLogger("de.edenmarket.test")`.

Ein Logger kann Ausgaben in 7 Stufen erzeugen:
SEVERE – WARNING – DEBUG – INFO – FINE – FINER
– FINEST

Entsprechend gibt es 7 Methoden `severe()`,
`warning()`,...

Logging

Standardmäßig werden nur Meldungen bis zur Stufe INFO auf System.err ausgegeben; dies ist in der globalen Konfigurationsdatei

`JRE_HOME/lib/logging.properties`

festgelegt.

Ausgegeben wird nur, wenn sowohl der Logger als auch sein Handler das entsprechende Level unterstützen.

Logging

Dies kann entweder global oder in den entsprechenden Objekten durch `setLevel` eingestellt werden.

Der Handler wird dem Logger durch `addHandler` zugeordnet:

Logging

```
log = Logger.getLogger("de.edenmarket.test");  
log.setLevel(Level.FINE);  
Handler handler1 = new ConsoleHandler();  
handler1.setLevel(Level.FINE);  
log.addHandler(handler1);
```

Um die Performance zu verbessern, kann man vor jeder log-Aktion abfragen, ob das entsprechende Level unterstützt wird:

```
if (log.isLoggable(Level.FINE)) {  
    log.fine(„Meldung“);  
}
```

Logging

Um die Logmeldungen in eine Datei zu schreiben, muß man einen FileHandler anlegen. Diesen kann man mit einem Patternstring anlegen. Die wichtigsten Abkürzungen sind:

- "/" Lokaler Pfadtrenner
- "%t" Temporäres Verzeichnis
- "%h" Homeverzeichnis
- "%g" Nummer bei Rotation
- "%%" Prozentzeichen

Logging

Zusätzlich kann man angeben:

- Maximale Größe in Bytes
- Zahl der Dateien bei Rotation (werden numeriert

`new FileHandler("%t/java%g.log", 1000000, 2)`
würde

z.B. nach `C:\TEMP\java0.log` und
`C:\TEMP\java1.log` schreiben.

Logging

Standardmäßig sind die Log-Records in Dateien als XML abgespeichert; der Formatierer kann durch `setFormatter` eingestellt werden.

Um dasselbe Layout wie auf der Konsole zu erhalten, kann man einen `SimpleFormatter` einsetzen:

```
handler2.setFormatter(new SimpleFormatter());
```

Übung 2, Aufgabe 2

Gliederung

1. Einleitung
2. Formate
3. Internationalisierung
4. Logging
5. *Preferences*
6. Innere Klassen
7. Reguläre Ausdrücke

Preferences

Unter „Präferenzen“ versteht man Einstellungen, die jeder einzelne Benutzer im Programm vornimmt, und die das Programm möglichst persistent speichern sollte, z.B. Anordnung von Fenstern.

In Eclipse erreicht man sie über den Menüpunkt Window- > Preferences; hier sind sie z.B. sehr umfangreich.

Preferences

Um Benutzereinstellungen abzuspeichern, könnte man ganz normale File-I/O implementieren. Strings könnten in einem Properties-Objekt, allgemeine Objekte in einer HashMap abgelegt werden.

Dies ist jedoch nicht immer vorteilhaft:

- Programm auf CD/DVD
- Programm aus jar-Archiv
- Keine Konvention für Dateinamen

Preferences

Jedes Betriebssystem kennt eine Standard-Speicherungsart; bei Windows ist das die Registry, also eine systemweite Datenbank mit separaten Einträgen für jeden Benutzer.

Java bietet seit dem JDK 1.4 eine Abstraktion dieses Betriebssystem- Services an.

Preferences

Man erhält den Einstieg über die `systemRoot` oder die `userRoot`, je nachdem, ob man die Einstellungen systemweit oder nur für den Benutzer vornehmen will:

```
Preferences root1 = Preferences.systemRoot();
```

```
Preferences root2 = Preferences.userRoot();
```

Von der Wurzel ausgehend kann man Pfade entlanggehen:

```
Preferences node = root2.node("/de/edenmarket");
```

Preferences

Meist baut man Preferences wie Packagenamen auf. Dann muß man nicht den Umweg über eine Wurzel gehen, sondern kann direkt den gewünschten Knoten herausfinden:

```
Preferences node = Preferences.userNodeForPackage(obj.getClass());
```

Dadurch erhält das Package seinen eindeutigen Knoten im Userbereich.

Preferences

Um Werte darin abzuspeichern (meist beim Beenden des Programms) gibt es ähnliche Methoden wie bei einer HashMap, allerdings nicht für Objekte, sondern für elementare Typen:

- putInt
- putDouble
- putByteArray
- ...

putByteArray könnte ein serialisiertes Objekt aufnehmen.

Preferences

Hier werden jeweils Werte mit einem String als Key versehen:

```
prefs.putInt("size", 240);
```

Zum Auslesen der Werte (meist beim Starten des Programms) gibt es entsprechende get-Methoden, die neben dem Key noch einen Defaultwert erfordern. Dadurch funktioniert das Programm auch, wenn es hier Probleme gibt.

Preferences

```
int size = prefs.getInt("size", 200);
```

Es ist möglich, einen Teilbaum oder einen einzelnen Knoten der Preferences zu exportieren und ihn woanders zu importieren. Dadurch können Einstellungen zwischen Rechnern übertragen werden:

- `exportSubtree(OutputStream out)`
- `exportNode(OutputStream out)`
- `importPreferences(InputStream in)`

Übung 2, Aufgabe 3

Gliederung

1. Einleitung
2. Formate
3. Internationalisierung
4. Logging
5. Preferences
6. *Innere Klassen*
7. Reguläre Ausdrücke

Innere Klassen

Eine innere Klasse ist eine Klasse, die innerhalb einer anderen definiert ist. Sie kann die 4 üblichen Zugriffsqualifizierer erhalten. Dadurch kann eine ganze Klasse vor anderem Code verborgen werden.

Es gibt insgesamt drei Varianten:

- „Normale“ innere Klassen
- Statische innere Klassen
- Lokale innere Klassen (= anonyme Klassen)

Innere Klassen

```
public class Person {
    private S tring name;
    private Adresse adresse;
    public Person(S tring name, Adresse adresse) {
        this.name =name;
        this.adresse =adresse;
    }
    private class Adresse {
        private S tring strasse, plz, ort;
        public Adresse(S tring strasse, S tring plz, S tring ort) {
            this.strasse =strasse;
            this.plz =plz;
            this.ort =ort;
        }
    }
}
```

Innere Klassen

Hier ist Adresse als private innere Klasse innerhalb von Person definiert. Somit kann außerhalb der Personklasse kein Code die Adresse verwenden.

Ein Adresse-Objekt erhält implizit immer ein umgebendes Person-Objekt und kann auch auf dessen private Elemente zugreifen. Folgendes wäre möglich:

Innere Klassen

```
public Adresse(String strasse, String plz, String ort) {  
    this.strasse = strasse;  
    this.plz = plz;  
    this.ort = ort;  
  
    System.out.println(name);  
        // name aus äußerer Klasse  
}
```

Innere Klassen

Dies wird dadurch realisiert, daß der Compiler ein zusätzliches Feld in jeden Konstruktor der inneren Klasse einfügt, so daß das Objekt der äußeren Klasse übergeben werden kann.

Die äußere Klasse kann in einer beliebigen Stelle Objekte der inneren Klasse erzeugen:

```
Adresse a =new Adresse("strasse", "plz", "ort");
```

Innere Klassen

Man kann aber auch in anderen Klassen Objekte einer nicht-privaten inneren Klasse erzeugen. Damit der Bezug zum äußeren Objekt hergestellt wird, muß folgende Syntax verwendet werden:

```
Person p = new Person();  
Person.Adresse adresse = p.new Adresse("", "", "");
```

Innere Klassen

Wenn man in der inneren Klasse das aktuelle Objekt der äußeren Klasse ansprechen will, muß man dem Schlüsselwort `this` den Klassennamen der äußeren Klasse voranstellen:

- `this` (innerhalb von Adresse) meint Adresse-Objekt
- `Person.this` (innerhalb Adresse) meint zugehöriges Person-Objekt

Innere Klassen

Manchmal will man eine Klasse innerhalb einer anderen kapseln, benötigt aber in der inneren Klasse keinen Bezug zu einem äußeren Objekt.

In diesem Fall sollte man das Schlüsselwort „static“ hinzufügen. Dann gibt es keine Notation „Person.this“ wie oben, da das innere Objekt kein zugeordnetes äußeres Objekt hat.

Dies ist eine statische innere Klasse (entspricht nested class in C++).

Innere Klassen

Das Erzeugen des inneren Objektes kann in der äußeren Klasse oder (bei nicht-private) auch anderswo erfolgen, in diesem Fall mit dem normalen new- Operator ohne vorangestelltes Objekt. Allerdings wird die innere Klasse immer noch mit dem zusammengesetzten Klassennamen bezeichnet:

```
Person.Adresse a = new Person.Adresse(...);
```

Übung 3, Aufgabe 1

Innere Klassen

Ein Spezialfall von inneren Klassen sind lokale Klassen; diese werden ohne Zugriffsqualifizierer innerhalb einer Methode definiert.

Lokale Klasse können als Unterklassen bestehender Klassen oder aber als Implementierungen bestehender Interfaces definiert werden:

Innere Klassen

```
WindowAdapter wa = new WindowAdapter() {  
    public void windowClosing(WindowEvent we) {  
        save();  
        System.exit(0);  
    }  
};
```

Hier wird eine Unterklasse von WindowAdapter mit überschriebener Methode windowClosing definiert. Stets wird ein Oberklassenkonstruktor aufgerufen (bei Interfaces immer Defaultkonstruktor).

Innere Klassen

Da man eine solche lokale Klasse nur einmal benötigt, erhält sie keinen Namen. Sie kann sogar innerhalb einer Anweisung definiert werden:

```
addWindowListener(new WindowAdapter() {  
    public void windowClosing(WindowEvent we) {  
        save();  
        System.exit(0);  
    }  
});
```

Innere Klassen

Beispiel mit Interface (bei Swing benötigt):

```
Runnable r = new Runnable() {  
    public void run() {  
        // doWork  
    }  
};
```

Hier müssen alle Methoden des Interface implementiert werden (bei Runnable nur eine).

Innere Klassen

Vorher definierte lokale Variable können nur dann angesprochen werden, wenn sie als `final` deklariert werden:

```
final int i = 10; // final ist notwendig  
Runnable r = new Runnable() {  
    public void run() {  
        // doWork verwendet i  
    }  
}
```

Übung 3, Aufgabe 2

Gliederung

1. Einleitung
2. Formate
3. Internationalisierung
4. Logging
5. Preferences
6. Innere Klassen
7. *Reguläre Ausdrücke*

Reguläre Ausdrücke

Reguläre Ausdrücke sind nötig, um in Strings Pattern zu finden und z.B. alle Email-Adressen oder Telephonnummern aus einem Text auszufiltern.

Seit dem **JDK 1.4** sind sie im package `java.util.regex` verfügbar; vorher gab es mehrere Open Source Lösungen (u.a. von Apache).

Reguläre Ausdrücke

Vorgehensweise: Man erzeugt mit Hilfe eines Patternstrings ein Pattern, dann mit dem zu untersuchenden String einen matcher. Die Methode matches zeigt, ob der String das Pattern realisiert.

```
String regex ="abc*";  
Pattern pattern =Pattern.compile(regex);  
String s ="abcdd";  
Matcher matcher =pattern.matcher(s);  
System.out.println(matcher.matches());
```

Reguläre Ausdrücke

In den Pattern kann man u.a. folgende Ausdrücke verwenden:

- Einzelne Zeichen abc
- Explizite Zeichenklassen [0-9], [A-Za-z], wobei der Bereich über Unicode gebildet wird
- Komplement: [^a-z]
- Vordefinierte Zeichenklassen: \d (Ziffern), \D (Keine Ziffer), \s (Leerraum), \S (Kein Leerraum), \w (Wortzeichen), \W (analog)

Reguläre Ausdrücke

- . jedes Zeichen, evt. ohne Zeilenende
- \ als Escapezeichen, also \. paßt für Punkt
- XY Konkatenation, X|Y Alternative
- Quantifizierer: X+ (mind. 1), X* (0 oder mehr), X? (0 oder 1)
- X{3} (dreimal), X{3,} (mind. dreimal), X{3,6} (zwischen dreimal und sechsmal)

X, Y stehen für reguläre Ausdrücke

Reguläre Ausdrücke

Der reguläre Ausdruck abc^* steht also für die Stringmenge $ab, abc, abcc, abccc, \dots$

Die Sprachen, die durch reguläre Ausdrücke definiert werden können (Typ-3-Sprachen, Kontextfreie Sprachen), sind in der theor. Informatik sehr gut untersucht. Sie gestatten noch keine „interessante“ Syntax (wie z.B. XML-Schemas), sind aber in der Praxis sehr wichtig.

Übung 4, Aufgabe 1

Reguläre Ausdrücke

Häufig ist man nicht nur daran interessiert, ob ein Pattern erfüllt wird, sondern auch, von welchen Teilstrings verschiedene Teile des Patterns realisiert werden.

Z.B. Könnte man einen Namen aufsplitten in (evt. mehrere) Vornamen und einen Nachnamen, durch Leerzeichen getrennt:

`(\w*\s)+\w*`

Reguläre Ausdrücke

In diesem Fall muß man jede gewünschte Gruppe klammern, und die vorhandenen Gruppen werden durch die Indizes 1, 2, ... bezeichnet und können über `matcher.group(1)` usw. abgefragt werden.

Hier werden also 2 zusätzliche Klammern benötigt:

`((\w*\s)+)(\w*)`, die Gruppe 1 steht für die Vornamen und Gruppe 3 für die Nachnamen.

Die Gruppe 2 `(\w*\s)` wird durch den letzten Vornamen realisiert.

Übung 4, Aufgabe 2