10.02

Javamagazin

Deutschland € 4,90 Österreich € 5,80
Schweiz SFr 9,60 / Luxemburg € 5,80

S&S

# Java™ magazin

Internet & Enterprise Technology

XML extra included

# IDEs für Java

## Rational XDE, Together ControlCenter, Borland JBuilder, Sun ONE Studio/NetBeans, AnyJ

## EJBs testen ohne Appserver

### Konzepte für schnellere Testphasen

### Reguläre Ausdrücke in Java
Offline-Reader mit regex API erstellen

### Eclipse: Grafik-Framework SWT
Performante GUIs mit dem Standard Widget Toolkit

### TINI: Java meets Hardware
GPS-Satellitendaten auswerten

### XML-basiertes EJB-Design
XML-Alternative für EAI-Anwendungen

### „Ich verwende selbst Together"
OO-Guru Peter Coad im Gespräch

mit CD!

www.javamagazin.de

D 45867

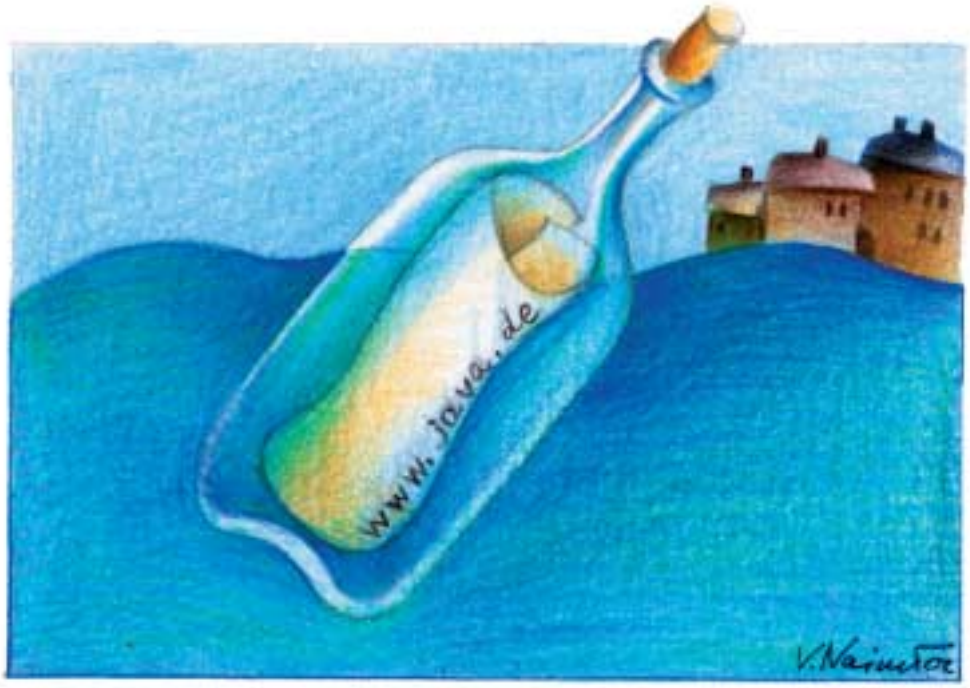
4 194586 704905 10

## Reguläre Ausdrücke in Java 1.4

von Marc-Oliver Scheele

# Textverarbeitung mit Java

Bereits seit den ersten Tagen von Java gab es externe Module, welche die Sprache um die Funktion von Regulären Ausdrücken erweiterten. Hierbei handelt es sich um ein sehr mächtiges Werkzeug, um Texte zu verarbeiten. Egal ob Texteingaben verifiziert werden, ob strukturierte Texte interpretiert oder ob Texte modifiziert werden müssen – die Regulären Ausdrücke sind hierfür die erste Wahl. Mit Java 1.4 wurden die Regulären Ausdrücke in das Standard API aufgenommen, womit sie jedem Java-Softwareentwickler zur Verfügung stehen. Dieser Artikel gibt einen Überblick in das neue *regex* API und zeigt den Einsatz exemplarisch anhand eines Offline-Readers.




Der Ursprung der Regulären Ausdrücke ist in den 50er Jahren festzumachen. Der Mathematiker Stephen Kleene führte so genannte *Regular Sets* ein, welche er als Notation in seinem Arbeitsgebiet der Logik nutzte. 1968 sprang die Erfindung dann durch Ken Thompson, ein Forscher in den Bell Labs, in die Informatik über. Thompson entwickelte einen auf Regulären Ausdrücken basierten Suchalgorithmus, der in den Unix-Editor ed integriert wurde. In der Folge blieb die Technologie insbesondere ein Leckerbissen für Tools im Unix-Umfeld. Gut bekannte *regex*-Kommandos sind z.B. *grep*, *sed*, *awk*, und *lex*.

Eine der wohl bekanntesten Programmiersprachen, die sich die Regulären Ausdrücke daraufhin einverleibt hat, ist die 1987 vorgestellte Sprache Perl. Heutzutage kann es sich keine professionelle Programmierumgebung mehr leisten, auf entsprechende Funktionalität zu verzichten. Im Java-Umfeld hat sich die letzten Jahre insbesondere das ORO-Modul durchgesetzt. Seit letztem Jahr ist das Projekt unter Verantwortung der Apache-Organisation [4]. Sowohl das *ORO*-Modul wie auch das neue *regex*-Modul in Java 1.4 sind im Großen und Ganzen zu den Regulären Ausdrücken in Perl 5 kompatibel.

### *regex* API

Das *java.util.regex*-Package besteht aus nur zwei Klassen und einer zugehörigen Exception. Dadurch ist das API trotz seiner Leistungsfähigkeit extrem übersichtlich. Abbildung 1 zeigt das zugehörige Klassendiagramm.

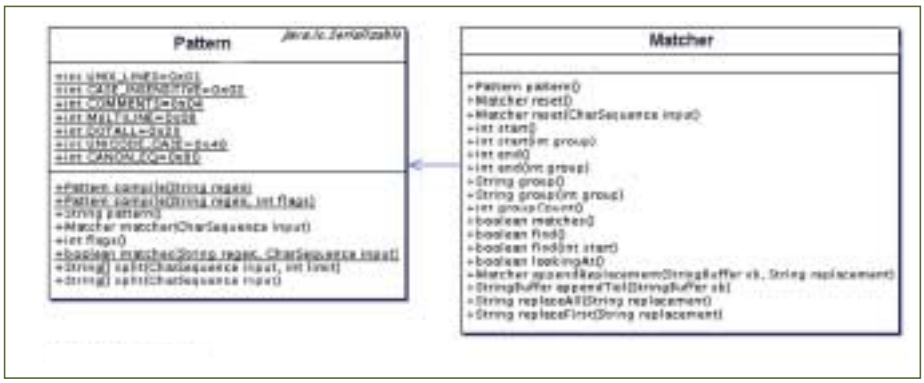



Abb. 1: Die zwei Klassen aus dem *java.util.regex*-Package

Ein Objekt der Klasse *Pattern* kapselt einen Regulären Ausdruck. Die Klasse selbst besitzt keinen Constructor. Stattdessen wird ein Objekt über die *compile()*-Factory-Methoden erzeugt. Bei der Objekt-Erzeugung wird das spezifizierte Pattern intern kompiliert. Später bei dem Vergleich zwischen Pattern und Text nutzt der Matching-Algorithmus ausschließlich das kompilierte Pattern, um die aufwändige Berechnung performant durchführen zu können.

Die zweite Klasse *Matcher* ist, wie es der Name vermuten lässt, für das Matching zwischen Pattern und Text zuständig. Das heißt sie stellt Methoden zur Verfügung, um das Pattern und den Text auf Gleichheit zu prüfen. Weiter erlaubt der *Matcher* die Extraktion von Textbestandteilen und die Modifikation des zu prüfenden Textes. Da ein *Matcher* seine Arbeit immer auf Basis eines spezifizierten Patterns durchführt, ist eine Instanz konsequenterweise nur über eine Factory-Methode eines *Pattern*-Objekts zu erzeugen: *public Matcher matcher (CharSequence input)*.

An der *matcher()*-Methode und anderen Methoden der *regex*-Klassen (Abb. 1) fällt auf, dass der zu überprüfende Input-Text nicht in Form eines String-Objekts spezifiziert wird, sondern ein Objekt vom Typ *CharSequence* verlangt wird. Hierbei handelt es sich um ein neues Interface in Java 1.4. Es beschreibt eine (nur) lesbare Sequenz von Zeichen und stellt eine Abstraktion für lesbare „Zeichen-Objekte“ dar. Implementiert wird diese Schnittstelle in Java 1.4 von der *String* und *StringBuffer*-Klasse. Darüber hinaus implementiert die *CharBuffer*-Klasse aus dem neuen *I/O*-Modul dieses Interface [1]. Durch diese Neuerung kann das Matching neben der String-Klasse direkt gegen weitere Klassen wie z.B. einen String-Buffer durchgeführt werden. Dies spart dem Java-Entwickler lästige Umwandlungen und kommt der Performance zugute.

## Kleine Beispiele

Schauen wir uns ein paar einfache Beispiele an, um die Funktionsweise des *regex* API zu verdeutlichen:

```
Pattern p = Pattern.compile("CDU.*FDP.*");
Matcher m = p.matcher("CDU SPD FDP PDS Grünen");
boolean b = m.matches();
```

In der ersten Zeile erzeugen wir das Pattern. Die Variable *p* enthält nun eine Referenz auf ein kompiliertes Pattern, welches einen Text beschreibt, der mit *CDU* anfängt und in der Folge irgendwo *FDP* enthält. Die Zeichen . und * innerhalb des Regulären Ausdrucks sind Sonderzeichen. Der Punkt repräsentiert die vordefinierte *Character-Class* für ein beliebiges Zeichen. Der Stern ist ein so genannter *Quantifier* und besagt, dass das vorstehende Zeichen null bis beliebig oft vorkommt.

In der zweiten Zeile erzeugen wir über die *factory*-Methode ein *Matcher*-Objekt. Das Argument beinhaltet den zu prüfenden Input-Text. Im letzten Schritt nutzen wir das frisch erzeugte *Matcher*-Objekt, um zu prüfen, ob der Reguläre Ausdruck mit dem Input-Text matched. Die Variable *b* enthält letztendlich den Wert *true*, da *CDU* und *FDP* vorhanden sind und die *CDU* vorne und vor der *FDP* steht.

| Construct | Matches | Example (/regex/ matches „text“ à group1) |
|---|---|---|
| **Characters** | | |
| x | The character x | /a/ „a“ |
| \\ | The backslash character | /\\/ „\“ |
| \t | The tab character ('\u0009') | /aaa\taaa/ „aaa aaaa“ |
| \n | The newline (line feed) character | /aaa\naaa/ „aaa aaa“ |
| \r | The carriage-return character | |
| | | |
| **Character classes** | | |
| [abc] | a, b, or c (simple class) | /a[xyz]a/ „aya“ |
| [^abc] | Any character except a, b, or c (negation) | /a[^xyz]a/ „aba“ |
| [a-zA-Z] | a through z or A through Z, inclusive (range) | /a[K-O]a/ „aLa“ |
| [a-z&&[def]] | d, e, or f (intersection) | /a[K-O&&[MXY]]a/ „aMa“ |
| [a-z&&[^bc]] | a through z, except for b and c: [ad-z] (subtraction) | /a[K-O&&[^MXY]]a/ „aLa“ |
| [a-z&&[^m-p]] | a through z, and not m through p: [a-lq-z](subtraction) | /a[K-O&&[^L-Z]]a/ „aKa“ |
| | | |
| **Predefined character classes** | | |
| . | Any character (may or may not match line terminators) | /a...a/ „aXYZa“ |
| \d | A digit: [0-9] | /a\d\d\da/ „a478a“ |
| \D | A non-digit: [^0-9] | /a\D\D\Da/ „aXYZa“ |
| \s | A whitespace character: [ \t\n\x0B\f\r] | /a\s\s\sa/ „a a“ |
| \S | A non-whitespace character: [^\s] | /a\S\S\Sa/ „aXYZa“ |
| \w | A word character: [a-zA-Z_0-9] | /a\w\w\wa/ „aX3Za“ |
| \W | A non-word character: [^\w] | /a\W\W\Wa/ „a.,>a“ |
| | | |
| **Boundary matchers** | | |
| ^ | The beginning of a line | /^abc/ „abc“ |
| $ | The end of a line | /abc$/ „abc“ |
| | | |
| **Greedy quantifiers** | | |
| X? | X, once or not at all | /Ab?b?/ „abb“ |
| X* | X, zero or more times | /A(.*)A.*/ „AxyzAabcA“ à xyzAabc |
| X+ | X, one or more times | /A(.+)A.+/ „AxyzAabcA“ à xyz |
| X{n} | X, exactly n times | /Ab{3}A/ „AbbbA“ |
| X(n,} | X, at least n times | /Ab{3,}A/ „AbbbbbA“ |
| X{n,m} | X, at least n but not more than m times | /Ab{2,4}A/ „AbbbbA“ |
| | | |
| **Reluctant quantifiers** | | |
| X?? | X, once or not at all | /Ab??b??/ „abb“ |
| X*? | X, zero or more times | /A(.*?)A.*?/ „AxyzAabcA“ à xyz |
| X+? | X, one or more times | /A(.+?)A.+?/ „AxyzAabcA“ à xyz |
| X{n}? | X, exactly n times | /Ab{3}?A/ „AbbbA“ |
| X(n,}? | X, at least n times | /Ab{3,}?A/ „AbbbbbA“ |
| X{n,m}? | X, at least n but not more than m times | /Ab{2,4}?A/ „AbbbbA“ |
| | | |
| **Logical operators** | | |
| XY | X followed by Y | /abc/ „abc“ |
| X\|Y | Either X or Y | /a\|b\|c/ „b“ |
| (X) | X, as a capturing group | |

| | | |
|---|---|---|
| **Back references** | | |
| \n | Whatever the nth capturing group matched | /(a.)\\1/ „abab“ |
| | | |
| **Quotation** | | |
| \ | Nothing, but quotes the following character | /\+/ „+“ |
| \Q | Nothing, but quotes all characters until \E | /\Q+.*?\E/ „+.*?“ |
| \E | Nothing, but ends quoting started by \Q | /\Q+.*?\E/ „+.*?“ |
| | | |
| **Special constructs (non-capturing)** | | |
| (?:X) | X, as a non-capturing group | |
| (?idmsux-idmsux) | Nothing, but turns match flags on - off | |
| (?idmsux-idmsux:X) | X, as a non-capturing group with the given flags on - off | |
| (?=X) | X, via zero-width positive lookahead | |
| (?!X) | X, via zero-width negative lookahead | |
| (?<=X) | X, via zero-width positive lookbehind | |
| (?<!X) | X, via zero-width negative lookbehind | |
| (?>X) | X, as an independent, non-capturing group | |

Tabelle: Zusammenfassung wesentlicher Elemente der Regulären Ausdrücke

Für Schnellschreiber bietet die API die Möglichkeit, obige Funktionalität in einem Ausdruck zu definieren:

```
boolean b = Pattern.matches("CDU.*FDP.*","CDU SPD FDP
                                        PDS Grünen");
```

Bei Ausführung dieses Ausdrucks werden intern exakt die drei Methodenaufrufe von oben durchgeführt. Das heißt, es handelt sich hierbei wirklich nur um eine „Schreibhilfe“ und um keine Performanceoptimierung. Im Zweifel ist dieser Ausdruck sogar langsamer: Wird der Reguläre Ausdruck mehrmals verwendet, so wird unter Gebrauch des Einzeilers das Pattern jedes Mal neu kompiliert. Über die einzelnen Methodenaufrufe wäre das kompilierte Pattern wiederverwendbar.

Was bieten die beiden Klassen noch für interessante Funktionalitäten? Schauen wir uns zunächst noch mal die *Pattern*-Klasse an. Erwähnenswert ist die Tatsache, dass man bei der Erzeugung eines *Pattern*-Objekts so genannte Flags mit angeben kann. Dabei handelt es sich um eine Bitmaske, über welche der Entwickler das Verhalten bei der Interpretation von Regulären Ausdrücken steuern kann. Beispielsweise ist in der Praxis häufig ein anderes Verhalten des Punkt-Sonderzeichens gewünscht. Standardmäßig steht der Punkt für jedes beliebige Zeichen, allerdings mit Ausnahme von Newline (*\n*). Will man obiges Parteienbeispiel dahin erweitern, dass ebenfalls ein Zeichenumbruch zwischen der *CDU* und *FDP* erlaubt ist, so würde man das Pattern wie folgt erzeugen:

```
Pattern p = Pattern.compile("CDU.*FDP.*",Pattern.DOTALL);
```

Sehr nützlich ist die Servicemethode *split()* eines *Pattern*-Objekts. Sie erlaubt das Spalten einer Zeichenkette durch Angabe eines Regulären Ausdruckes. Beispiel:

```
Pattern p = Pattern.compile(";|#");
String[] result = p.split("CDU=37%;SPD=33%#FDP=18%;
                                    Grüne=6%#PDS=5%");
```

Hier wird die angegebene Zeichenkette an dem Zeichen *;* oder # geteilt. Das Ergebnis ist ein Array der Länge fünf mit folgendem Inhalt:

```
[0]=CDU=37% [1]=SPD=33% [2]=FDP=18% [3]=Grüne=6%
                                    [4]=PDS=5%
```

Vielseitiger sind die Funktionalitäten der *Matcher*-Klasse. Neben der Prüfung, ob ein Regulärer Ausdruck mit einem Text matched (übereinstimmt), wird in der Praxis sicherlich von der Rückgabe gefundener Teilzeichenketten am häufigsten Gebrauch gemacht. Hierzu können innerhalb des Regulären Ausdrucks durch Klammersetzung so genannte Groups (Gruppen) definiert werden. Mehrere Gruppen eines Regulären Ausdrucks werden von links nach rechts sequentiell durchnummeriert, sodass ein gezielter Zugriff möglich ist. Die Gruppe 0 steht generell für den gesamten Ausdruck. Im folgenden Beispiel ist die Aufgabe, den Teilstring zwischen CDU und FDP zu extrahieren:

```
Pattern p = Pattern.compile("CDU(.*)FDP.*");
Matcher m = p.matcher("CDU SPD FDP PDS Grünen");
```

Ein Aufruf von *m.group(1)* ergibt die Zeichenkette *SPD*. Ein Aufruf von *m.group(0)* bzw. von *m.group()* liefert den gesamten Ausdruck zurück: *CDU SPD FDP PDS Grünen*

Bei der *group()*-methode handelt es sich im übrigen um eine Serviceroutine, deren vollständige Funktion durch die *start()*- und *end()*-Methoden des *Matchers* realisiert werden könnte. Unter der Annahme, dass die Variable *s* für unseren Parteien-String steht, ist der folgenden Ausdruck wahr:

```
m.group(1).equals(s.substring(m.start(1), m.end(1)))
```

Möchte man ein *Matcher*-Objekt wiederverwenden, so kann die *reset()*-Methode aufgerufen werden. Anschließend sind sämtliche Statusinformationen des *Matchers* zurückgesetzt, so als ob das Objekt frisch von dem zugehörigen *Pattern*-Objekt erzeugt worden wäre. Weiter bietet die *Matcher*-Klasse Methoden, um direkt den Input-Text zu modifizieren. Die entsprechenden *replace()*- und *append()*-Methoden werden weiter unten anhand des Code-Beispiels erläutert.

## *regex*-Ausdrücke

Die Mächtigkeit des *regex* API entsteht durch die richtige Anwendung von Regulären Ausdrücken. Hierzu ist es notwendig, dass sich der Softwareentwickler einen fundierten Überblick über Syntax und Semantik von Regulären Ausdrücken erarbeitet.

### OfflineLoader – ein Java-Open Source-Projekt

Die Java-basierte Software OfflineLoader ist ein Download-Manager für Web-Seiten (auch als Offline-Reader bezeichnet). Die Kernfunktion dieser Software liegt darin, HTML-Seiten inklusive aller eingebeteten Elemente (z.B. Bilder) lokal verfügbar zu machen. Hierdurch können Web-Sites zum einen offline betrachtet werden und zum anderen kann ein Informationsstand dauerhaft archiviert werden.
Das Projekt verfolgt zwei Ziele:

1. Die Software soll ein nützliches Tool für den Internet-Surfer bieten und somit ein erfolgreiches „Open Source-Geschenk" an die Internetgemeinschaft darstellen. Folgende wesentliche Funktionen sind zur Erreichung des Ziels in der Entwicklung:

- professionelle Kernfunktionen eines Download-Managers (insbesondere wichtige Optionen zum Downloadverhalten)
- timergesteuerte Downloads
- differenzielle Downloads, die nur die Änderungen seit dem letzten Download berücksichtigen
- mehrerer Outputformate, z.B. HTML, ZIP oder Konvertierung nach PDF
- Verwaltung, Katalogisierung und Volltextsuche über alle Downloads

2. Die Software soll einen weiteren Beweis dafür antreten, daß Java im Tool- und Client-Umfeld erfolgreich eingesetzt werden kann. Hierfür wird auf moderne Schnittstellen gesetzt:

- Einfache grafische Swing-basierte Oberfläche
- Lauffähig über die Kommandozeile
- Deployment über die Web Start-Technologie
- Servicefunktionalitäten werden als Web Services angeboten und veröffentlicht

Es handelt sich bei dem OfflineLoader um ein junges Projekt, welches Mitte dieses Jahres gestartet wurde. Weitere Java-Softwareentwickler sind herzlich eingeladen sich an der weiteren Entwicklung zu beteiligen. Information: www.moscon.de/offlineloader/

Eine ausführliche Einführung in dieses Thema würde den Rahmen dieses Artikels sprengen. Stattdessen findet sich in der Tabelle eine Übersicht der wichtigsten Elemente von Regulären Ausdrücken. Anhand der Beispiele in der Tabelle kann der Laie einen fundierten Überblick gewinnen und der Profi hat eine Referenz zum schnellen Nachschlagen an der Hand.

## Vielseitigkeit der Matcher-Klassen

Im Wesentlichen entspricht die Mächtigkeit der Regulären Ausdrücke in Java 1.4 denen von der Programmiersprache Perl 5. Folgende wenige und in der Java-Umgebung wohl kaum notwendige Konstrukte werden in Perl, nicht aber in Java unterstützt:

- Konditionale Konstrukte der Form *(?{X})* und *(?(condition)X|Y)*
- Ein Konstrukt, welches die Integration von Programmcode erlaubt: *(?{code})* und *(??{code})*
- Ein integrierter und nicht interpretierter Kommentar der Form *(?#comment)*
- Die Preprozessor-Operationen *\l \u \L \U*, welche den Ausdruck in Groß- bzw. Kleinbuchstaben umwandeln

Zur weiteren Vertiefung in das Thema Reguläre Ausdrücke sei im übrigen das sehr gute Buch „Mastering Regular Expressions“ von Jeffrey E.F. Friedl empfohlen [2].

### Ein Offline-Reader

Abschließend wollen wir den Einsatz der Regulären Ausdrücke unter Java anhand eines Praxisbeispiels betrachten. Bei dem Beispiel handelt es sich um einen als Open Source veröffentlichten Offline-Reader. Diese Software dient dazu, Web-Seiten lokal herunterzuladen, um sie beispielsweise auch ohne Internet-Zugang vollständig lesen zu können. Für weitere Informationen zu dem speziellen Open Source-Projekt siehe den Kasten „Offline-Loader – ein Java Open Source-Projekt“ und [3].

Die Kernfunktion eines Offline-Readers lässt sich wie folgt zusammenfassen:

- Suche alle Hyperlinks und eingebetteten Elemente in einem Startdokument;
- Lade alle eingebetteten Elemente (z.B. Bilder) herunter;
- Lade die verlinkten Dokumente herunter;
- Modifiziere alle Links, sodass sie auf die heruntergeladenen Dokumente und Elemente zeigen;
- Wiederhole den Vorgang für die verlinkten Dokumente.

Obige Kernfunktion zeigt, daß ein Offline-Reader also hauptsächlich Textstellen finden und modifizieren können muss. Hierfür ist natürlich die Mustererkennung über Reguläre Ausdrücke prädestiniert.

Listing 1 zeigt den relevanten Ausschnitt der entsprechenden *Parser*-Klasse. Im Folgenden eine kurze Erläuterung hierzu: Zu Beginn werden die Regulären Ausdrücke zum Auffinden von *<a href>*- und *<img src>*-Tags definiert. Sie sind so flexibel spezifiziert, daß beliebige Leerzeichen, Attribute und auch das Nichtvorhandensein von Anführungszeichen nicht weiter stören. Zu beachten sind die doppelten Quotes bei den Sonderzeichen (doppelter Backslash z.B. \\s). Dies ist notwendig, um die Interpretation im Vorfeld durch den Java-Compiler zu verhindern.

Die Methode *parse()* iteriert durch die definierten Ausdrücke und prüft pro Durchlauf das Dokument auf ein *Pattern*. In der Methode *checkForPattern()* geht es dann ans Eingemachte: Wir erzeugen ein *Pattern*- und ein *Matcher*-Objekt, um dann über ein *matcher.find()* unser *Pattern* im Text zu finden. Auffällig ist hier, daß wir zum Matchen nicht die bereits oben beschriebene Methode *matches()* nutzen. Der Unterschied zwischen *matches()* und *find()* liegt darin, dass erstere Methode sich auf den gesamten Text bezieht, während ein *find()* Teilbereiche des Textes untersucht. Würden wir also hier *matches()* benutzen, so müsste das gesamte Dokument dem definierten Link-Pattern entsprechen.

**Listing 1**

```
....

/**
 *  Praxisbeispiel aus dem Open-Source Projekt
                                        "OfflineLoader"
 *  http://www.moscon.de/offlineloader/
 */
public class ParserHtml implements Parser {

 // Reguläre Ausdrücke für HTML Anchor- & Link-Tags
 //
 static private DownloadType[] types =
                                    new DownloadType[2];
 static {
  types[0] = new
DownloadType("<\\s*?a.+?href\\s*?=\\s*?(?:\"|')?
                        (\\S+?)(?:\".*?>|'.*?>|\\s.*?>|>)",
            "LINK: <a>",DownloadType.HTML_TEXT);
  types[1] = new
DownloadType("<\\s*?img.+?src\\s*?=\\s*?(?:\"|')?
                        (\\S+?)(?:\".*?>|'.*?>|\\s.*?>|>)",
            "IMAGE: <img>", DownloadType.BINARY);
 }

 .....

 // Start-Methode, um in einem HTML-Dokument nach
                                  obigen Pattern zu suchen
 //
 public void parse() throws DownloadException {
  if (orgDoc==null)
   throw new DownloadException("No Document was set
                              for parsing!  (orgDoc==null)");
  localDoc=orgDoc;
  for (int i= 0; i < types.length; i++) {
   localDoc = checkForPattern(types[i],localDoc);
  }
 }

 .....

 // Durchsucht das Dokument nach einem Link-Pattern
                                  und modifiziert den Link
 //
 private String checkForPattern (DownloadType type,
                                   String orgDocument) {
  Pattern pattern = Pattern.compile(type.getPattern(),
        Pattern.CASE_INSENSITIVE+Pattern.DOTALL+
                Pattern.MULTILINE+Pattern.UNIX_LINES);
  Matcher matcher = pattern.matcher(orgDocument);
  StringBuffer modDoc = new StringBuffer();
  String replacement;

  while (matcher.find()) {
   replacement=calcAbsLink(matcher);
  else
   replacement=calcLocalLink(matcher);
  matcher.appendReplacement(modDoc,replacement);
 }
 matcher.appendTail(modDoc);

 return modDoc.toString();
}


// Wandelt relative Links in absolute URLs
//
private String calcAbsLink(Matcher matcher) {
 if (baseURL==null) {
  System.err.println("baseURL not set!!!");
  return matcher.group();
 }
 String docURL = matcher.group(1);
 String absURL;

 // check for abs-server-urls
 if (docURL.startsWith("http://"))
  return matcher.group();
 // build prefix
 String prefix = "http://"+baseURL.getHost();
 if (baseURL.getPort()>0) {
  prefix += ":"+baseURL.getPort();
 }
 // check for absolute-path
 if (docURL.startsWith("/")) {
  absURL = prefix+docURL;
 } else {
  // relative-path
  int pos;
  if (baseURL.getFile() != null &&
(pos=baseURL.getFile().lastIndexOf('/'))>-1)
    prefix = prefix+baseURL.getFile().substring(0,pos);
  absURL = prefix+"/"+docURL;
 }

 StringBuffer repla = new StringBuffer();
 repla.append(matcher.group().substring(0,matcher.start
                                        (1)-matcher.start()));
 repla.append(absURL);
 repla.append(matcher.group().substring(matcher.end
                                        (1)-matcher.start()));
 return repla.toString();
}

 ........

}
```

Liefert der *find()*-Aufruf *true*, so nutzen wir die *matcher.appendReplacement()*-Methode, um den gefundenen Link zu modifizieren (z.B. in eine absolute URL). Hierzu übergibt man der Methode einen *StringBuffer*, der mit dem Teilstring des Input-Texts vom letzten bis zum aktuellen Match gefüllt wird. Der übergebene Replacement-String ersetzt den Match; in unserem Fall wird also das gefundene HTML *A*- oder *IMG*-Tag ersetzt. Nachdem kein weiteres *Pattern* gefunden wurde, wird mittels *matcher.appendTail()* der *StringBuffer* mit dem restlichen Input-Text gefüllt. Im Ergebnis beinhaltet der *StringBuffer* den originalen Input-Text, wobei alle Links durch absolute URLs ausgetauscht worden sind.

Die Methode *calcAbsLink()* dient dazu, aus einem relativen Link den absoluten zu berechnen. Hier kann der Einsatz der *group(), start(), end()* eines *Matcher*-Objekts beobachtet werden.

## Fazit

Die Entscheidung, die Java Plattform 1.4 um ein Standard API für Reguläre Ausdrücke zu erweitern, ist als sehr positiv zu bewerten. Heutzutage sollte jede ausgereifte Programmierplattform eine entsprechende Funktionalität anbieten. Das Verarbeiten von Texten gehört schließlich zu den wesentlichen Funktionen der meisten Softwareprojekte.

Begrüßenswert ist ebenfalls die sehr übersichtliche *regex* API. Da sie nur aus zwei schlanken Klassen besteht, erhält der in Regulären Ausdrücken bereits geschulte Java-Entwickler eine mächtige Funktion, für dessen Nutzung er nur eine sehr geringe Einarbeitungszeit benötigt. ■

*Marc-Oliver Scheele ist freiberuflich tätiger Diplom-Informatiker, der seine Kunden u.a. in Java-Entwicklungsprojekten unterstützt (scheele@moscon.de).*

### Links & Literatur

[1] I/O und Netzfunktionen: Das N im I/O, *Java Magazin* 6.2002, Seite 16

[2] Mastering Regular Expressions, Jeffrey E. F. Friedl, O'Reilly and Associates, 1997

[3] Open-Source Projekt „OfflineLoader": www.moscon.de/offlineloader/

[4] Apache ORO: jakarta.apache.org/oro/index.html