Gottlieb-Daimler-Schule 2
Technisches Schulzentrum Sindelfingen
mit Abteilung Akademie für Datenverarbeitung

Inhaltsverzeichnis

1 Einführung

Das Durchsuchen von Texten nach bestimmten Wörtern oder Zeichenmustern (engl. Pattern) ist eine Standardaufgabe in der Programmierung. Möchte man z.B. alle E-Mail Adressen suchen, die in einer Webseite vorkommen, dann benötigt man eine Möglichkeit, den generellen Aufbau (das Zeichenmuster) einer E-Mail Adressen zu beschreiben. Zur Definition von Zeichenmustern gibt es eine eigene formale Sprache. Diese Metasprache nennt sich „reguläre Ausdrücke“ und wurde von dem Mathematiker Stephen Cole Kleene entwickelt. Eine Besonderheit dieser Metasprache besteht darin, dass es möglich ist, aus einem definierten Zeichenmuster (dem regulären Ausdruck) automatisch einen Algorithmus zu konstruieren. Die Java Bibliothek besitzt, wie viele andere moderne Programmiersprachen, entsprechende Methoden, um reguläre Ausdrücke nutzen zu können.

2 Beispiele für reguläre Ausdrücke und ihre Anwendung

2.1 Überprüfen einer MAC-Adresse auf Korrektheit

```
    public static void main(String[] args)
    {
      String sMacAdr; // Eingabe-Zeichenkette
      // Regulärer Ausdruck (Regex) für MAC-Adr:
1     String sRegExMac = "([0-9A-Fa-f]{2}\\-){5}[0-9A-Fa-f]{2}";
      // Generiere aus dem "regulären Ausdruck" den "Matcher"-Algorithmus
2     Pattern matchAlgorithm = Pattern.compile(sRegExMac);

      sMacAdr = Eingabe.getString("Enter MAC Adr: ");

      // Verbinde den "Matcher"-Algorithmus mit der Eingabe Zeichenkette
3     Matcher matcher = matchAlgorithm.matcher(sMacAdr);
      // Entspricht die Zeichenkette dem Zeichenmuster?
4     if (matcher.matches())  // Wenn Zeichenmuster -> true
        System.out.println("Adr. o.k.");
      else
        System.out.println("Adr. nicht o.k.");
    }
```

1	Das Zeichenmuster („Pattern“) einer MAC-Adresse kann durch einen „regulären Ausdruck exakt beschrieben werden.
2	Aus dem regulären Ausdruck (Regex = „regular expression“) kann der Algorithmus zum Auffinden von Zeichenketten, die dem Zeichenmuster entsprechen, automatisch generiert werden („Matcher“-Algorithmus)
3	Die Eingabe-Zeichenkette wird mit Matcher-Algorithmus verknüpft
4	`match()` sucht in der Eingabe-Zeichenfolge nach dem nächsten Vorkommen des Zeichenmusters. Falls die Zeichenkette als ganzes dem Zeichenmuster genügt, liefert `match()` `true` zurück, sonst `false`

Wird, wie im Beispiel der MAC Adressüberprüfung, der Matcher-Algorithmus nur einmalig angewendet, so können mehrere Teilschritte zu einem zusammengefasst werden; zu diesem Zweck kann die Klassenmethode `Pattern.matches(…)` verwendet werden.

```
    public static void main(String[] args)
    {
      String sMacAdr; // Eingabe-Zeichenkette
      // Regulärer Ausdruck (Regex) für MAC-Adr:
1     String sRegExMac = "([0-9A-Fa-f]{2}\\-){5}[0-9A-Fa-f]{2}";

2     sMacAdr = Eingabe.getString("Enter MAC Adr: ");
      // Generiere aus dem "regulären Ausdruck" den "Matcher"-Algorithmus
      // Verbinde den "Matcher"-Algorithmus mit der Eingabe Zeichenkette
      // und prüfe, ob die Zeichenkette dem Zeichenmuster entspricht?

3     if (Pattern.matches(sRegExMac, sMacAdr))
        System.out.println("Adr. o.k.");
4     else
        System.out.println("Adr. nicht o.k.");
    }
```

2.2 Finden von MAC-Adressen in einer Zeichenkette

```
    public static void main(String[] args)
    {
      boolean found = false;
      String sMacAdr; // Eingabe-Zeichenkette
1     String sRegExMac = "([0-9A-Fa-f]{2}\\-){5}[0-9A-Fa-f]{2}";
      // Generiere aus dem "regulären Ausdruck" den "Matcher"-Algorithmus
2     Pattern matchAlgorithm = Pattern.compile(sRegExMac);
        // Zeichenkette aus einer DHCP Konfigurationsdatei
      sMacAdr = "A205A;00-56-71-36-AF-15;192.168.75.110;6;7.12.2019"
               +"A205B;00-56-71-36-B3-B2;192.168.75.111;6;7.12.2019"
               +"A205C;00-56-89-41-00-00;192.168.75.112;6;7.12.2019";

3     Matcher matcher = matchAlgorithm.matcher(sMacAdr);
      // Suchen nach MAC Adressen in der Zeichenkette
4     while (matcher.find()) {
        System.out.printf("\nI found the text \"%s\" starting at " +
          "index %3d and ending at index %3d.%n",
5         matcher.group(), matcher.start(), matcher.end());
        found = true;
      }
      if(!found){
          System.out.printf("No match found.%n");
      }
    }
```

Ausgabe:

```
I found the text "00-56-71-36-AF-15" starting at index   6 and ending at index  23.
I found the text "00-56-71-36-B3-B2" starting at index  56 and ending at index  73.
I found the text "00-56-89-41-00-00" starting at index 106 and ending at index 123.
```

1	Das Zeichenmuster („Pattern“) einer MAC-Adresse kann durch einen „regulären Ausdruck exakt beschrieben werden.
2	Aus dem regulären Ausdruck (Regex = „regular expression“) kann der Algorithmus zum Auffinden von Zeichenketten, die dem Zeichenmuster entsprechen, automatisch generiert werden („Matcher“-Algorithmus)
3	Die Eingabe-Zeichenkette wird mit Matcher-Algorithmus verknüpft
4	`find()` sucht in der Eingabe-Zeichenfolge nach dem nächsten Vorkommen des Zeichenmusters. Falls in der Zeichenkette ein Teilstring vorkommt, der dem Zeichenmuster genügt, liefert `find()` `true` zurück, sonst `false`
5	`group()` liefert den durch `find()` gefundenen Teilstring. `start()` liefert den Startindex des gefundenen Teilstrings und `end()` entsprechenden Endindex.

3 Einführung in die Syntax regulärer Ausdrücke (RegexSyntax)

Um alle möglichen Fälle für Zeichenmuster abzudecken, benötigt man auch eine entsprechend umfangreiche Syntax für die Formulierung von regulären Ausdrücken. Diese komplett hier zu behandeln, würde den Rahmen sprengen. Wenn man die Syntax nicht häufig anwendet, vergisst man sie auch relativ schnell wieder, deshalb ist es sinnvoller sich bei konkretem Bedarf intensiver damit zu beschäftigen. Da die Anwendung von regulären Ausdrücken aber in der Praxis von großer Bedeutung ist, sollen hier die Grundlagen gelegt werden.

3.1 String Literale (wortwörtlicher Text) in regulären Ausdrücken

Ein Zeichenmuster kann auch wortwörtlichen Text (Literale) enthalten.
Bsp.:

```
  String sText = "Hallo Welt!"; // Eingabe-Zeichenkette
  // Regulärer Ausdruck bestehend aus Literal:
  String sRegEx = "Welt";

  if (Pattern.compile(sRegEx).matcher(sTest).find())
    System.out.println("gefunden"); // "Welt" kommt in "Hallo Welt!" vor
  else
    System.out.println("nicht gefunden");
}
```

Achtung: Kommen in dem gesuchten wortwörtlichen Text sogenannte Metazeichen vor, so müssen diese mit einem '\' maskiert werden. Metazeichen sind solche Zeichen, die in der Regex Syntax eine besondere Bedeutung haben.

Die Metazeichen, die in Java Bibliothek unterstützt werden sind folgende:
`<([{\^-=$!|]})?*+.>`

Bsp.:

```
Pattern.compile("Welt.").matcher("Hallo Welt!").find() // → true
// "Welt." kommt in "Hallo Welt!" nicht vor, wird aber trotzdem gefunden
// weil der Punkt ein Metazeichen ist, dass für ein beliebiges Zeichen steht
Pattern.compile("Welt\\.").matcher("Hallo Welt!").find() // → false
```

Möchte man in einem regulären Ausdruck innerhalb eines Literals auch Steuerzeichen verwenden, so werden diese gleich dargestellt, wie wir das bereits von normalen Stringliteralen kennen:

`\\`	The backslash character
`\t`	The tab character (`'\u0009'`)
`\n`	The newline (line feed) character (`'\u000A'`)
`\r`	The carriage-return character (`'\u000D'`)
`\f`	The form-feed character (`'\u000C'`)
`\a`	The alert (bell) character (`'\u0007'`)
`\e`	The escape character (`'\u001B'`)

3.2 Zeichenklassen in regulären Ausdrücken

In einem regulären Ausdruck kann man festlegen, dass an einer bestimmten Stelle im Zeichenmuster nur bestimmte Zeichen stehen dürfen. Dazu verwendet man Zeichenklassen.

Vordefinierte Zeichenklassen

.	Ein beliebiges Zeichen
\d	Beliebiges Ziffernzeichen; identisch mit [0-9]
\D	Alle Zeichen außer Ziffern; identisch mit [^0-9]
\s	Leerraumzeichen (White Space); identisch mit [\t\n\x0B\f\r]
\S	Alle Zeichen außer Leerraum; identisch mit [^\s]
\w	Jedes alphanummerisches Zeichen; identisch mit [a-zA-Z_0-9]
\W	Alle Zeichen außer Jedes alphanummerische; identisch mit [^\w]

Bsp.: (zur Klassenmethode `Pattern.matches` s.o. 2.1)

```
if (Pattern.matches("\\d\\d\\d", "120")) // → true
  System.out.println("Zeichenkette besteht aus 3 Ziffern");
else
  System.out.println("falsch");
```

Bsp.:

```
Pattern.matches("\\d\\d\\d", "20")) // → false, es werden 3 Dezimalziffern erwartet
```

Selbstdefinierte Zeichenklassen

[abc]	a, b oder c, (Einfache Klasse)
[^abc]	Alles außer a, b oder c (Negierung)
[a-zA-Z]	a bis z oder A bis Z (einfacher Zeichenbereich)
[a-m[n-p]]	a bis m oder n bis p; identisch mit [a-mm-p] bzw. [a-p] (Vereinigung)
[a-z&&[^bc]]	a bis z, ohne b und ohne c (Zeichenbereich mit Ausschluss)

3.3 Wiederholungen von Zeichen(klassen) in regulären Ausdrücken (Quantitätsangaben)

In einem regulären Ausdruck kann man festlegen, dass an einer bestimmten Stelle im Zeichenmuster ein bestimmtes Zeichen, ein Zeichen einer Zeichenklasse oder ein Teilmuster mehrfach stehen darf. (X steht für ein beliebiges Teilmuster (Teilausdruck))

Quantitäten

$X?$	X genau einmal oder überhaupt nicht
$X*$	X 0 Mal bis viele
$X+$	X 1 Mal bis viele
$X\{n\}$	X genau n Mal
$X\{n,\}$	X mindestens n Mal bis viele
$X\{n,m\}$	X n bis m Mal

Bsp.: (Alternative Lösung zum vorletzten Bsp.)

```
if (Pattern.matches("\\d{3}", "120")) // → true
  System.out.println("Zeichenkette besteht aus 3 Ziffern");
else
  System.out.println("falsch");
```

Bsp. `split`-Methode:

```
sZeilenTeile = sZeile.split("[ \t]+");
```

Der Parameter der `split`-Methode ist ebenfalls ein regulärer Ausdruck und im Bsp. folgendermaßen zu interpretieren:

`[ \t]` → selbstdefinierte Zeichenklasse: Leerzeichen oder Tab-Zeichen

`[ \t]+` → Quantität +: Zeichenmuster beschreibt eine beliebige Folge von Leer- und Tab-Zeichen, mindestens aber ein Zeichen

3.4 Aneinanderreihung und Gruppenbildung in regulären Ausdrücken

Aneinanderreihung und Gruppierung

XY	X gefolgt von Y
$X\|Y$	Entweder X oder Y
(X)	X als Gruppe mit späterer Bezugsmöglichkeit
$(?:X)$	X als Gruppe ohne späterer Bezugsmöglichkeit

- Eine Zeichenfolge erfüllte den regulären Ausdruck XY wenn sein Anfang X erfüllt und sein hinterer Teil Y
- Eine Zeichenfolge erfüllte den regulären Ausdruck $X|Y$ wenn er X erfüllt oder Y

Bsp.:

```
Pattern.matches("Hallo (Welt )+", "Hallo Welt Welt ")) // → true
Pattern.matches("Hallo (Welt )+", " Hallo Welt Hallo Welt ")) // → false
```

- Soll ein Wiederholungszeichen nur auf einen Teilausdruck angewendet werden, so kann man die Gruppenbildung verwenden: z.B. $X(Y)+$ → + bezieht sich nur auf Y