Sinds versie 1.4 zijn Regular Expressions (kortweg RegEx) onderdeel van Java. Deze kunnen veel code terugbrengen tot slechts enkele statements. In het eerste deel uit een serie van twee artikelen van Jesse Houwing en Peter van den Berkmortel wordt niet alleen naar de Java-classes gekeken waarmee RegEx in Java is geïmplementeerd, maar vooral ook naar de RegEx syntax zelf. Aan de hand van praktijkvoorbeelden worden de praktische toepassingen van RegEx toegelicht en duidelijk gemaakt.

Regular Expressions in Java (1)

Herkenning van patronen

Tijdens de Masters of Java 2005 konden deelnemers een opgave maken over reguliere expressies in Java. Reguliere expressies (kortweg RegEx) zijn expressies om patroonherkenning op teksten te doen. Een RegEx beschrijft een patroon, en een tekst voldoet daar geheel, gedeeltelijk of niet aan. RegEx worden door veel talen ondersteund. Vanaf versie 1.4 worden ze standaard ondersteund in de Java Runtime Environment. Voor die tijd waren alleen third party implementaties voor Java beschikbaar.

De Java-implementatie van RegEx draait in eerste instantie om het java.util.regex package met daarin de classes Pattern en Matcher. Pattern representeert een gecompileerde reguliere expressie. Matcher interpreteert het Pattern en biedt functionaliteit om match- (en replace-) operaties uit te voeren. Wat het voordeel van RegEx is, is het beste uit te leggen met een voorbeeld.

```
// Valideer een postcode
Pattern pattern =
  Pattern.compile(
    "(?!0)\\d{4}[ ]?[A-Z]{2}");

boolean isPostcode(String pc)
{
  if (pc == null) return false;

  Matcher m = pattern.matcher(pc);
  return m.matches();
}
```

We zien dat het hier gaat om slechts vier regels code. Hoe de RegEx-syntax er uit ziet en hoe Matcher precies te werk gaat, zien we later. Laten we deze code eerst vergelijken met een klassieke postcode-controle. Hier zien we duidelijk wat de voordelen van RegEx zijn.

```
// Valideer een postcode
String ALFA = "ABCDEFGHI..XYZ";
String NUM = "0123456789";

boolean isPostcode(String pc)
{
  boolean hasSpatie = false;
  boolean result = pc != null;

  for (int i = 0;
       result && i < pc.length();
       i++)
  {
    switch (i)
    {
      case 0:
        result = pc.charAt(i)
             != '0';
      case 1: // FALLTHROUGH
      case 2: // FALLTHROUGH
```

```
      case 3:
        result &= NUM.indexOf(
            pc.charAt(i)) > -1;
        break;
      case 4:
        if (pc.charAt(i) == ' ')
        {
          hasSpatie = true;
          break;
        }
        // FALLTHROUGH
      default:
        result = ALFA.indexOf(
          pc.charAt(i)) > -1;
    }
  }

  result &= hasSpatie ?
    pc.length() == 7 :
    pc.length() == 6;

  return result;
}
```

Toepassingsgebieden RegEx worden veel toegepast bij een aantal veel voorkomende problemen. Het valideren van (gebruikers-) input bijvoorbeeld. Zoals uit het voorbeeld duidelijk wordt, zijn RegEx erg goed in het valideren van bijvoorbeeld een postcode. Maar ook datums, integer of drijvende kommagetallen en alfabetische letters kunnen goed met RegEx worden gecontroleerd. Ook voor het parsen van teksten of programmacode wordt RegEx veelvuldig gebruikt. Programmacode wordt tijdens het compilen meestal met RegEx gevalideerd en geparsed. Het voordeel is duidelijk: de code voor het valideren wordt hergebruikt voor het parsen. De hoeveelheid code die geschreven moet worden blijft beperkt en is bovendien (gedeeltelijk) opnieuw te gebruiken. RegEx wordt ook gebruikt voor het selecteren van bepaalde informatie uit een grotere tekst. De RegEx die controleert of een invoertekst een e-mail adres is, kan ook gebruikt worden om te controleren of een langere tekst een e-mail adres bevat en om dat e-mail adres te selecteren (na een kleine aanpassing). Het belangrijkste voordeel van het gebruik van RegEx is dat er weinig programmacode nodig is om complexe zaken te controleren. Ook al groeit de complexiteit, de hoeveelheid code blijft vaak nagenoeg gelijk.

Matcher Hoe gaat de Matcher nu precies te werk? Matcher kan op verschillende manieren trachten om een match te vinden voor een bepaalde input en een Pattern. In de postcode-validatie wordt gebruik gemaakt van de `matches` methode. Deze probeert de volledige input te matchen met het Pattern (van begin tot eind). Met de `lookingAt` methode wordt geprobeerd de input te matchen vanaf het begin. De `find` methode zoekt telkens naar de volgende match. We zullen later de `lookingAt` en `find` methodes nader bekijken, maar eerst tonen we hoe een en ander in zijn werk gaat. De RegEx voor het postcode voorbeeld ziet er als volgt uit: `[1-9]\d{3}[ ]?[A-Z]{2}`.

Het eerste deel (`[1-9]`) geeft aan dat er een karakter moet komen in de range 1 tot en met 9. De rechte blokhaken geven een keuze aan. Het minteken geeft een range aan. Het tweede deel (`\d{3}`) geeft aan dat er drie cijfers moeten komen. `\d` is een zogenaamde Character Class, in dit geval een afkorting voor een cijfer (digit). Het aantal (`{3}`) slaat altijd op het element dat voor de beginaccolade staat.

- *In de Java-code staat een extra backslash voor de `\d`. Dit is omdat een backslash voor de Java String class een speciaal karakter is en dus ge-escapet moet worden.*

Vervolgens een optionele spatie (`[ ]?`). De keuze wordt beperkt tot één karakter (spatie). Het vraagteken dat volgt geeft aan dat het een optioneel element is. De keuzehaken zijn in dit geval niet noodzakelijk, maar maken het Pattern wel veel duidelijker. Bovendien zijn de keuzehaken in andere situaties wel degelijk noodzakelijk. Het laatste deel van de expressie (`[A-Z]{2}`) geeft een range van A tot Z, met een aantal van 2.

Het Pattern wordt als instance-variabele gedeclareerd, zodat het kan worden hergebruikt. Hergebruik van Matcher is ook mogelijk, maar wordt niet aangeraden: Matcher is namelijk niet Thread-safe (Pattern wel).

RegEx Elementen Laten we nu eens kijken naar de elementen in een RegEx. Elk normaal karakter die in een RegEx voorkomt, moet ook in de input voorkomen. Dus:

```
// Voorbeeld Normale Karakters
Pattern: abc
Match: abc, maar niet bbc.
```

Speciale karakters

\\	Backslash
\t	Tab
\n	New line
\r	Carriage return
\f	Form feed
\xhh	Hexadecimale waarde
\uhhhh	Hexadecimale waarde

- *Als je een speciaal karakter in je code opneemt, moet deze nog eens extra ge-escapet worden. Dus een backslash in je code ziet er zo uit:* \\\\

```
// Voorbeeld Speciale Karakters
Pattern: a\tb
Match: a<tab>b, maar niet a b.
<< einde kader met computercode >>
```

De Character Classes geven keuzes aan. Er zijn verschillende vormen:

[abc]	Keuze: de waarden a, b of c
[^abc]	Negatie: een waarde ongelijk aan a, b en c
[a-z]	Range: de waarden a tot en met z
[a-zA-Z]	Inclusie: de waarden a tot en met z en A tot en met Z.
[a-d[f-z]]	Unie: a tot en met d en f tot en met z; zelfde als [a-df-z]
[a-z&&[def]]	Intersectie: de waarden d, e of f
[a-z&&[^def]]	Subtractie: de waarden a tot en met z, maar niet d, e of f
[a-z&&[^d-g]]	Subtractie: de waarden a tot en met z, maar niet d tot en met g

- *Inclusie, unie, intersectie en subtractie zijn alleen mogelijk in de Java Regular Expression API. Andere RegEx implementaties staan dit niet toe.*
- *Als het minteken aan het begin of het einde van de tekenreeks voorkomt, dan wordt aangenomen dat het minteken bij de mogelijke keuzes hoort.*

Er zijn ook een aantal voorgedefinieerde Character Classes:

.	Elke waarde
\d	Digit: [0-9]
\D	Geen digit: [^0-9]
\s	Witruimte: [\t\n\x0B\f\r]
\S	Geen witruimte: [^\s]
\w	Een woord karakter: [a-zA-Z_0-9]
\W	Geen woord karakter: [^\w]

- *De Character Classes zijn gedefinieerd op de Unicode tekenset. De exacte definities van de Character Classes kunnen daarom verschillen naar gelang het Locale waaronder gewerkt wordt! In veel gevallen is het daarom te prefereren om [0-9] te gebruiken in plaats van* \d.

```
// Voorbeeld Character Classes
Pattern: [a-z][A-Z]\s\d
Match: aA<spatie>4 en aB<spatie>4, maar niet
aA<spatie>Z (geen cijfer aan het eind).

Pattern: \w[/|-]
Match: 7- en a/, maar niet A! (! komt niet in
de keuze lijst voor).
```

RegEx kent ook wildcards. Deze vallen onder de categorie quantifiers. Quantifiers geven aan hoe vaak een bepaald deel van de RegEx voor mag komen in de input.

Greedy quantifiers

?	nul of een keer
*	nul of meer keer
+	een of meer keer
{n}	voorgaande element n keer
{n,}	voorgaande element n of meer keer
{n,m}	voorgaande element minimaal n en maximaal m keer

Deze quantifiers worden 'greedy' genoemd, omdat ze zoveel mogelijk karakters proberen te matchen. Alleen als er anders geen match gemaakt kan worden, wordt (door middel van backtracking) het aantal karakters beperkt. Bij backtracking wordt telkens als een Pattern geen match oplevert een deel van de match ongedaan gemaakt (voor zover toegestaan volgens het RegEx element). Vervolgens wordt opnieuw geprobeerd om met het hele Pattern een match te maken. Dit wordt herhaald totdat een match gevonden is of totdat alle geldige mogelijkheden voor een RegEx element zijn uitgeput. Het ongedaan maken van een match gebeurt telkens met één karakter per keer. Later bekijken we nog andere soorten quantifiers. Eerst bekijken we groepering en alternatieven:

Ook voor het parsen van teksten of programmacode wordt RegEx veelvuldig gebruikt

(..)	Capturing group
(?:..)	Non-capturing group
|	Alternation (or)
\n	Back reference to capturing group #n

De karakters uit de input die met de RegEx element in een capturing-groep gematcht worden, kunnen bij een match van het hele Pattern via de Matcher worden opgehaald (met de `group(int)` methode). Bij een non-capturing groep kan dit niet. Het doel van non-capturing groepen is om RegEx elementen die bij elkaar horen en waarvoor verschillende alternatieven zijn, bij elkaar te zetten. Er kan niet via back-reference naar verwezen worden. Ze tellen dus ook niet tegen de `groupCount` (het aantal matches in een capturing group). Een back-reference dient altijd exact dezelfde karakters te matchen als de capturing group waarnaar verwezen wordt. Een back-reference refereert dus niet aan de RegEx-elementen in het Pattern, maar naar de input-karakters in de match.

```
// Voorbeeld Groepering
// en Quantifiers
Pattern: \d{2}([-/])\d{2}\1\d{4}
Match: 01-02-2006 en 99/14/2999, maar niet
01-02/2006 (want / is ongelijk aan -).

Pattern: (?:0[1-9])|(?:[12]\d{2})| (?:3[01])
Match: 09, 10, 29, maar niet 32.
```

Naast back-reference kan groepering worden gebruikt voor het selecteren van specifieke delen van een tekst. Het volgende voorbeeld parset regels met formaat {<param-list>:<code>} en maakt hiermee een Construct (een fictieve klasse). De constructor van Construct verwacht een array van argumenten en een codeblok (String).

```
// Voorbeeld Groepering en Code
Pattern p =
  Pattern.compile(
    "\\{(\\w+)(?:[,](\\w+))*"
  + "[:](.*)\\}");

Construct parse(String line)
{
  Matcher m = p.matcher(line);

  if (!m.matches())
    throw new ParseException();

  Object[] args =
    new Object[m.groupCount()-1];

  for (int i = 1;
       i < m.groupCount();
       i++)
  {
    args[i – 1] = m.group(i);
  }
  return new Construct(args,
    m.group(m.groupCount()));
}
```

Het gebruik van RegEx heeft hier als voordeel dat het controleren van geldigheid en het overslaan van boilerplate code (accolades, dubbele punten) vrijwel geen extra code kost. Het Pattern zelf (`\{(\w+)(?:[,](\w+)) *[:](.*)\}`) ziet er in eerste instantie vrij complex uit. Het eerste deel (`\{`) maakt een match met een accolade. Omdat een accolade ook een speciaal karakter is, is een escape (in de meeste gevallen) noodzakelijk. De eerste parameter wordt geselecteerd met `(\w+)`. Er moet minimaal één karakter geselecteerd worden (lege parameterlijsten zijn niet toegestaan). Er staan ronde haken omheen, zodat de waarde ertussen wordt gecaptured. De parameter telt dus tegen de groupCount en kan met de group(int) methode worden opgevraagd. De overige parameters (optioneel) worden geselecteerd met `(?:[,](\w+))*`. De eerste drie karakters (`(?:`) geven aan dat het een non-capturing groep is. Het groeperen is noodzakelijk om met de asterisk aan het einde aan te kunnen geven dat dit onderdeel nul of meer keer mag voorkomen. Verder is een komma verplicht (`[,]`) en natuurlijk de parameter zelf (`(\w+)`) in een capture groep, zodat deze weer telt tegen de groupCount.

Vervolgens moet een dubbele punt in de regel voorkomen (`[:]`). Alles tussen de dubbele punt en de sluitaccolade wordt met `(.*)\}` gevangen en geselecteerd. Het voorbeeld kan eenvoudig worden aangepast om witruimte toe te staan. Voor de leesbaarheid is dit in het voorbeeld weggelaten.

LookingAt en Find De `lookingAt` methode werkt vrijwel gelijk aan de `matches` methode. `matches` probeert een match te maken met de gehele input, terwijl `lookingAt` hetzelfde doet met een deel van de input.

```
// Voorbeeld LookingAt
Pattern: (\w+)\s\1
Looking At: aabc abca (het deel abc abc is
een match), maar niet aabc dabc (omdat de
back reference een exacte herhaling van de
input karakters moet zijn).
```

De `find` methode werkt net zoals de `lookingAt` methode, echter de positie waar de match is gevonden wordt bewaard, zodat `find` herhaaldelijk kan worden aangeroepen om de input sequentieel te doorzoeken.

```
// Voorbeeld find
Pattern p =
  Pattern.compile("\\w+");
```

```
Matcher m = p.matcher("Dit is een zin, die
doorzocht moet worden.");

while (m.find())
{
  System.out.println(m.group());
}
```

De `group()` methode (zonder argumenten) geeft altijd alle input karakters terug die (deze keer) gematcht zijn.

Boundary Matchers De volgende RegEx elementen zijn zogenaamde boundary matchers:

^	Begin van een string (voor de lookingAt methode)
$	Einde van een string (voor de lookingAt methode)
\b	Word boundary (begin of einde van een woord)
\B	Non-word boundary (niet het begin of einde van een woord)
\A	Begin van de input
\G	Einde van de vorige match (voor de find methode)
\Z	Einde van de input, met uitzondering van de laatste regel terminator
\z	Einde van de input

Deze elementen zorgen ervoor dat een RegEx verankerd wordt. Er kan verankerd worden aan de start en einde van de input, maar ook aan woorden en niet woorden.

- *Aanroep van* `lookingAt` *met een RegEx die begint met* ^ *en eindigt met* $ *is gelijk aan gebruik van de* `matches` *methode zonder deze toevoegingen.*

```
// Voorbeeld Boundary Matchers
Pattern: \b\w{3}
Input: Dit is een zin, die doorzocht moet
worden.
Find: (achtereenvolgens)
  Dit een zin die doo moe wor
```

In dit voorbeeld wordt een boundary-matcher gebruikt om telkens de eerste drie karakters van een woord terug te krijgen (voor woorden met drie of meer karakters). Als de word-boundary niet wordt gebruikt, worden veel meer karakters teruggegeven.

```
// RegEx die woorden met drie
// of minder letters selecteert
Pattern: \b\w{1,3}\b
Input: Dit is een zin, die doorzocht moet
worden.
Find: (achtereenvolgens)
  Dit is een zin die
```

Laten we nu naar de andere quantifiers kijken. Er zijn nog twee soorten: reluctant en possessive quantifiers.

Reluctant quantifiers

??	nul of een keer
*?	nul of meer keer
+?	een of meer keer
{n}?	voorgaande element n keer
{n,}?	voorgaande element n of meer keer
{n,m}?	voorgaande element minimaal n en maximaal m keer

We hebben gezien dat een greedy quantifier zoveel mogelijk karakters probeert te matchen. Een reluctant quantifier probeert zo min mogelijk karakters te matchen.

Possessive quantifiers

?+	nul of een keer
*+	nul of meer keer
++	een of meer keer
{n}+	voorgaande element n keer
{n,}+	voorgaande element n of meer keer
{n,m}+	voorgaande element minimaal n en maximaal m keer

Deze quantifiers proberen altijd zoveel mogelijk te matchen, net als de greedy quantifiers. Het verschil is, dat er met deze quantifiers geen backtracking plaats vindt, zodat er mogelijk geen match gevonden wordt (waar de andere quantifiers dat wel zouden doen).

```
// Voorbeeld Quantifiers
// Matches staan tussen kleiner en
//   groter dan tekens.
Input: Dit is een mooie zin, zeg!

Pattern (greedy): \b.+
Matches: <Dit is een mooie zin, zeg!>
Pattern (reluctant): \b.+?
Matches: <D> < > <i> < > <e> < > <m> < > <z>
<,> <z> <!>
Pattern (possessive): \b.++
Matches: <Dit is een mooie zin, zeg!>
```

```
Pattern (greedy): .+\b
Matches: <Dit is een mooie zin, zeg>
Pattern (reluctant): .+?\b
Matches: <Dit> < > <is> < > <een> < > <mooie>
< > <zin> <, > <zeg>
Pattern (possessive): .++\b
Matches:
```

Bovenstaand voorbeeld maakt duidelijk dat je goed moet nadenken over je RegEx. Omdat met `.++\b` eerst de gehele input opgeslokt wordt, kan er alleen een match gemaakt worden als de input eindigt met een word boundary (wat niet het geval is). Met `\b.++` krijgen we wel een match, omdat het eerste karakter van de input `D` is, dus het begin van de input is wel een word boundary. Het voorbeeld maakt ook het verschil tussen greedy en possessive duidelijk: waar greedy zoveel mogelijk probeert te matchen, rekening houdend met de rest van de RegEx, zal possessive zoveel mogelijk proberen te matchen, zonder rekening te houden met de rest van de RegEx.

Let ook op de reluctant quantifiers: als de word boundary (alleen) aan het begin staat, bestaat de match telkens uit slechts één karakter. Maar de word boundary aan het einde zorgt voor langere matches. Dit komt doordat de match wel vanaf het begin van de input (of de vorige match) gemaakt wordt. Bedenk dit: het minimale aantal karakters vanaf een word boundary (met een of meer) is natuurlijk een `(1)`; het minimale aantal karakters tot een word boundary, is de lengte van het eerstvolgende woord.

Voor possessive quantifiers geldt juist: het maximaal aantal karakters vanaf een word boundary is de hele zin; het maximaal aantal karakters tot een word boundary is de hele zin, slechts wanneer de zin eindigt met een word boundary. Als de zin niet eindigt met een word-boundary, dan is er geen match.

In het volgende deel uit deze serie zullen we kijken naar look-arounds, switches, Character Classes voor Unicode ondersteuning en het vervangen van tekst.

Referenties

JavaDoc
http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/package-summary.html
http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Character.UnicodeBlock.html

Alternatieve RegEx implementaties
http://sourceforge.net/projects/jregex
http://jakarta.apache.org/regexp/index.html

RegEx bibliotheken
http://regexlib.com/Default.aspx
http://www.regular-expressions.info/

RegEx testers
http://www.eclipse-plugins.info/eclipse/ plugin_details.jsp?id=964
http://brosinski.com/regex/
http://www.roblocher.com/technotes/regexp.aspx
http://www.regexbuddy.com/index.html

Jesse Houwing (e-mail: jesse.houwing @sogeti.nl) is werkzaam als Microsoft .NET Senior Application engineer bij Sogeti Nederland BV. Hij is de maker van de RegEx Fundamentals cursus die binnen Sogeti gegeven wordt.

Peter van den Berkmortel (e-mail: peter.vanden.berkmortel@sogeti.nl) is werkzaam als Technisch Architect in het Java Delivery Center van Sogeti. Hij is onder andere Certified JBoss Developer.