Pattern Matching with Regular Expressions

Dr. Subrat Kumar Nayak
Associate Professor
Department of CSE
ITER, SOADU

Regular Expression

- Regular expressions, or regexes for short, provide a concise and precise specification of patterns to be matched in text.

Example:

- Suppose you have a bunch of 150000 mail in your drive, And let's further suppose that you remember that somewhere in there is an email message from someone named Angie or Anjie. Or was it Angy? But you don't remember what you called it or where you stored it. Obviously, you have to look for it.
- Simplest way is to write a regular expression to search it:

An[^ dn].*

finding words that begin with "An", while the cryptic [^ dn] requires

The "An" to be followed by a character other than (^ means not in this context) a space (to eliminate the very common English word "an" at the start of a sentence) or "d" (to eliminate the common word "and") or "n" (to eliminate Anne, Announcing, etc.).

Subexpression	Matches	Notes
General		
\^	Start of line/string	
$	End of line/string	
\b	Word boundary	
\B	Not a word boundary	
\A	Beginning of entire string	

Subexpression	Matches	Notes
\p{Punct}	Punctuation characters	One of !"#$%&'()*+,-./:;<=>?@[]\^_`{|}\~
\p{Lower}	Lowercase characters	[a-z]
\p{Upper}	Uppercase characters	[A-Z]
\p{XDigit}	Hexadecimal digit characters	[0-9a-fA-F]

Subexpression	Matches	Notes
\z	End of entire string	
\Z	End of entire string (except allowable final line terminator)	See Recipe 4.9
.	Any one character (except line terminator)	
[…]	"Character class"; any one character from those listed	
[\^…]	Any one character not from those listed	See Recipe 4.2
Alternation and Grouping		
(…)	Grouping (capture groups)	See Recipe 4.3
|	Alternation	
(?:_re_)	Noncapturing parenthesis	
\G	End of the previous match	
\ *n*	Back-reference to capture group number "*n*"	
Normal (greedy) quantifiers		
{ *m*,*n* }	Quantifier for "from *m* to *n* repetitions"	See Recipe 4.4
{ *m* ,}	Quantifier for "*m* or more repetitions"	
{ *m* }	Quantifier for "exactly *m* repetitions"	See Recipe 4.10
{,*n* }	Quantifier for 0 up to *n* repetitions	
*	Quantifier for 0 or more repetitions	Short for {0,}
+	Quantifier for 1 or more repetitions	Short for {1,}; see Recipe 4.2
?	Quantifier for 0 or 1 repetitions (i.e., present exactly once, or not at all)	Short for {0,1}
Reluctant (non-greedy) quantifiers		
{ *m*,*n* }?	Reluctant quantifier for "from *m* to *n* repetitions"	
{ *m* ,}?	Reluctant quantifier for "*m* or more repetitions"	
{,*n* }?	Reluctant quantifier for 0 up to *n* repetitions	
*?	Reluctant quantifier: 0 or more	
+?	Reluctant quantifier: 1 or more	See Recipe 4.10
??	Reluctant quantifier: 0 or 1 times	
Possessive (very greedy) quantifiers		
{ *m*,*n* }+	Possessive quantifier for "from *m* to *n* repetitions"	
{ *m* ,}+	Possessive quantifier for "*m* or more repetitions"	
{,*n* }+	Possessive quantifier for 0 up to *n* repetitions	
*+	Possessive quantifier: 0 or more	
++	Possessive quantifier: 1 or more	

Subexpression	Matches	Notes
?+	Possessive quantifier: 0 or 1 times	
Escapes and shorthands		
\	Escape (quote) character: turns most metacharacters off; turns subsequent alphabetic into metacharacters	
\Q	Escape (quote) all characters up to \E	
\E	Ends quoting begun with \Q	
\t	Tab character	
\r	Return (carriage return) character	
\n	Newline character	See Recipe 4.9
\f	Form feed	
\w	Character in a word	Use \w+ for a word; see Recipe 4.10
\W	A nonword character	
\d	Numeric digit	Use \d+ for an integer; see Recipe 4.2
\D	A nondigit character	
\s	Whitespace	Space, tab, etc., as determined by java.lang.Character.isWhitespace()
\S	A nonwhitespace character	See Recipe 4.10
Unicode blocks (representative samples)		
\p{InGreek}	A character in the Greek block	(Simple block)
\P{InGreek}	Any character not in the Greek block	
\p{Lu}	An uppercase letter	(Simple category)
\p{Sc}	A currency symbol	
POSIX-style character classes (defined only for US-ASCII)		
\p{Alnum}	Alphanumeric characters	[A-Za-z0-9]
\p{Alpha}	Alphabetic characters	[A-Za-z]
\p{ASCII}	Any ASCII character	[\x00-\x7F]
\p{Blank}	Space and tab characters	
\p{Space}	Space characters	[\t\n\x0B\f\r]
\p{Cntrl}	Control characters	[\x00-\x1F\x7F]
\p{Digit}	Numeric digit characters	[0-9]
\p{Graph}	Printable and visible characters (not spaces or control characters)	
\p{Print}	Printable characters	

Regular Expression

Differences Among Greedy, Reluctant, and Possessive Quantifiers

- Greedy quantifiers are considered "greedy" because they force the matcher to **read in**, or *eat*, **the entire input string prior to attempting the first match. If the first match attempt** (the entire input string) **fails**, **the matcher backs off the input string by one character and tries again, repeating the process until a match is found or there are no more characters left to back off from.** Depending on the quantifier used in the expression, the last thing it will try matching against is 1 or 0 characters.
- The reluctant quantifiers, however, take the opposite approach: **They start at the beginning of the input string, then reluctantly eat one character at a time looking for a match.** The last thing they try is the entire input string.
- Finally, the possessive quantifiers always eat the entire input string, trying once (and only once) for a match. Unlike the greedy quantifiers, possessive quantifiers never back off, even if doing so would allow the overall match to succeed.

Regular Expression

Differences Among Greedy, Reluctant, and Possessive Quantifiers

Example:

Enter your regex: .*foo // **greedy quantifier**

Enter input string to search: xfooxxxxxx**foo**

I found the text "xfooxxxxxxfoo" starting at index 0 and ending at index 13.

Enter your regex: .*?foo // **reluctant quantifier**

Enter input string to search: **xfoo**xxxxx**xfoo**

I found the text "xfoo" starting at index 0 and ending at index 4.

I found the text "xxxxxxfoo" starting at index 4 and ending at index 13.

Enter your regex: .*+foo // **possessive quantifier**

Enter input string to search: xfooxxxxxx**foo**

No match found. (Failed to backtrack. Hence, **foo** seems to be missing for possesive)

Regular Expression

Greedy	Reluctant	Possessive	Meaning
X?	X??	X?+	X, once or not at all
X*	X*?	X*+	X, zero or more times
X+	X+?	X++	X, one or more times
X{n}	X{n}?	X{n}+	X, exactly *n* times
X{n,}	X{n,}?	X{n,}+	X, at least *n* times
X{n,m}	X{n,m}?	X{n,m}+	X, at least *n* but not more than *m* times

Enter your regex: a?
Enter input string to search:
I found the text "" starting at index 0 and ending at index 0.
Enter your regex: a*
Enter input string to search:
I found the text "" starting at index 0 and ending at index 0.
Enter your regex: a+
Enter input string to search:
No match found.

Regular Expression

Q. Write a regular expression to print all the name from n name start with Angie, Anjie or Angy.

Ans: An[^ nd].* //An[^ nd]+ Angelina will not match

Q. Write a regular expression to print the string from bunch of string starting with "A" followed by any number of character.

Ans: A.*/A.+

Q. Write a regular expression to find a match that starts with an alphabate and end with a digit.

Ans: \b\p{Alpha}{1,4}\d{1}\b

Q. Write a regular expression to find a match that starts with a vowel.

Ans: \b[aeiou]\p{Alnum}{1,}

Q. Write a regular expression to find a match that starts with a vowel and end with a vowel.

Ans: \b[aeiou]([0-9] | [a-z]){1,4}[aeiou]\b or

\b[aeiou]\p{Alnum}{1,9}[aeiou]\b or

\b([aeiou] | [AEIOU])\p{Alnum}{1,}[aeiou]\b

Regular Expression

Q. Write a regular expression that matches with a string that starts with a name like Angie/Anjie/Angy.

Ans: ^An[^ dn].*

Q. Write a regular expression that validates a date in MM/DD/YYYY format. Note: ignore leap year

Ans: ^(1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])/[0-9]{4}$

Q. Write a regular expression to find a match that starts with an uppercase letter and end with a digit.

Ans: \b\p{Upper}{1}\p{Alpha}{1,}\d{1}\b

Regular Expression

Using regexes in Java: Test for a Pattern

Matching regex using matches() in String class:

- If all you need is to find out whether a given **regex matches a string**, you can use the convenient boolean matches() method of the String class, which accepts a regex pattern in String form as its argument.

Example:

```
if ( inputString . matches ( stringRegexPattern )) {
      // it matched ... do something with it ...
}
```

Regular Expression

Java Regex:

- The **Java Regex** or Regular Expression is an API to *define a pattern for searching or manipulating strings*.

Matching regexes using Pattern and Matcher(s)

- If the regex is going to be used more than once or twice in a program, it is more efficient to construct and use a Pattern and its Matcher (s).
- The normal steps for regex

1. Create a Pattern by calling the static method Pattern.compile().
2. Request a Matcher from the pattern by calling pattern.matcher(CharSequence) for each String (or other CharSequence) you wish to look through.
3. Call (once or more) one of the finder methods (discussed later) in the resulting Matcher .

Regular Expression

java.util.regex package

- The Matcher and Pattern classes provide the facility of Java regular expression.

The Matcher class

It is a *regex engine* which is used to perform match operations on a character sequence.

The Matcher methods

- **boolean matches()**

Used to compare the **entire string** against the pattern; this is the same as the routine in java.lang.String .

- **lookingAt()**

Used to match the pattern **only at the beginning** of the string.

- **boolean find()**

Used to match the pattern in the string (not necessarily at the first character of the string), starting at the beginning of the string or, if the method was previously called and succeeded, at the first character not matched by the previous match.

Regular Expression

Pattern class

It is the *compiled version of a regular expression*. It is used to define a pattern for the regex engine.

Methods

- static Pattern compile(String regex)

compiles the given regex and returns the instance of the Pattern.

- Matcher matcher(CharSequence input)

creates a matcher that matches the given input with the pattern.

…

Regular Expression

matches()

```
import java . util . regex .*;
public class RESimple {
        public static void main ( String [] argv ) {
                String pattern = "pqr .*";
                String input ="pqr abd pxy ";
                Pattern p = Pattern . compile ( pattern );
                Matcher m=p. matcher ( input );
                if(m. matches ())
                {
                        System .out. println (" Patern "+ pattern +" found in string "+ input );
                }
                else
                {
                        System .out. println (" Patern "+ pattern +" not found in string "+ input );
        }}
}
```

Output : Patern pqr .* found in string pqr abd pxy

Regular Expression

lookingAt()

```
import java . util . regex .*;
public class RESimple {
        public static void main ( String [] argv ) {
                String pattern = "pqr ";
                String input ="pqr abd pxy ";
                Pattern p = Pattern . compile ( pattern );
                Matcher m=p. matcher ( input );
                if(m. lookingAt ())
                {
                        System .out. println (" Patern "+ pattern +" found in string "+ input );
                }
                else
                {
                        System .out. println (" Patern "+ pattern +" not found in string " + input );
        }}
}
```

Output : Patern pqr found in string pqr abd pxy

Regular Expression

find()

```
import java . util . regex .*;
public class RESimple {
        public static void main ( String [] argv ) {
                String pattern = "abd ";
                String input ="pqr abd pxy ";
                Pattern p = Pattern . compile ( pattern );
                Matcher m=p. matcher ( input );
                if(m. find ())
                {
                        System .out. println (" Patern "+ pattern +" found in string "+ input );
                }
                else
                {
                        System .out. println (" Patern "+ pattern +" not found in string "+ input );
        }}
}
```

Output : Patern abd found in string pqr abd pxy

Regular Expression

Finding the Matching Text

- You need to find the text that the regex matched with.

Related functions

start(), end()

- Returns the character position in the string of the starting and ending characters that matched.

groupCount()

- Returns the number of parenthesized capture groups **in the expression/regex**, if any; returns 0 if no groups were used.

group(int i)

- Returns the characters matched by group i of the current match, if i is greater than or equal to zero and less than or equal to the return value of groupCount().
- Group 0 is the entire match, so group(0) (or just group()) returns the entire portion of the input that matched.

Regular Expression

groupCount() example

```
import java . util . regex .*;
      public class RESimple {
            public static void main ( String [] argv ) {
            String pattern = " (.*) (\\d {6}) ";
            //Two groups
            String input =" abdpxy 100000 ";
            Pattern p = Pattern . compile ( pattern );
            Matcher m=p. matcher ( input );
            System .out. println (" Total group = "+m. groupCount ());
      }
}
```

Output : Total group =2

Regular Expression

```
import java . util . regex .*;
public class RESimple {
        public static void main ( String [] argv ) {
                String pattern = " (.*) (\\d {6}) ";
                String input =" abdpxy 100000 ";
                Pattern p = Pattern . compile ( pattern );
                Matcher m=p. matcher ( input );
                if(m. find ()){
                        System .out. println (" Patern "+ pattern +" found in string "+ input +" with group
                                "+m. group (0) );
                                                        // abdpxy 100000
                        System .out. println (" Patern "+ pattern +" found in string "+ input +" with group
                                "+m. group (1) );
                                                        // abdpxy
                        System .out. println (" Patern "+ pattern +" found in string "+ input +" with group
                                "+m. group (2) );
                                                        // 100000
                }
}}
```

Regular Expression

Replacing the Matched Text

- **replaceAll(newString)**

Replaces all occurrences that matched with the new string.

Example:

```
import java . util . regex . Pattern ;
import java . util . regex . Matcher ;
public class ReplaceAll {
        public static void main ( String args [])
        {
                String patt = "\\ bfavor \\b"; // A test input .
                String input = "Do me a favor ? Fetch my favorite .favor ";
                System .out. println (" Input : " + input );
                // Run it from a RE instance and see that it works
                Pattern r = Pattern . compile ( patt );
                Matcher m = r. matcher ( input );
        System .out. println (" ReplaceAll : " + m. replaceAll (" favour "));
}}
```

Regular Expression

- **appendReplacement(StringBuffer, newString)**

Copies up to before the first match, plus the given newString .

- **appendTail(StringBuffer)**

Appends text after the last match (normally used after appendReplacement).

```
public class ReplaceAll {
        public static void main ( String args [])
        {
                String patt = "\\ b favor \\b"; // A test input .
                String input = "Do me a favor ? Fetch my favorite (favor) ";
                System .out. println (" Input : " + input );
                Pattern r = Pattern . compile ( patt ); // Run it from a RE instance and see that it works
                Matcher m = r. matcher ( input );
                StringBuffer sb= new StringBuffer ();
                while (m. find ()) {
                m. appendReplacement (sb , " favour ");// Copy to before first match , plus the word " favor "
                }
                m . appendTail (sb); // copy remainder (comment this line to check the importance of appendTail)
                System .out. println (sb. toString ());
}}
```

Regular Expression

Pattern.compile() Flags

- **CASE_INSENSITIVE**

Turns on case-insensitive matching

Ex. Pattern reCaseInsens = Pattern . compile (pattern, **P**attern . CASE_INSENSITIVE)

// check the previous example using "Favor" instead of "favor".

- **COMMENTS**

Causes whitespace and comments (from # to endofline) to be ignored in the pattern.

- **DOTALL**

Allows dot (.) to match any regular character or the newline, not just any regular character other than newline.

- **MULTILINE**

Specifies multiline mode.

Task (Explore)

- **UNICODE_CASE**
- **UNIX_LINES**

Regular Expression

```
import java . util . regex .*;
public class NewLine
{
        public static void main ( String args [])
        {
                String input = "I dream of engines \ nmore
                engines , all day long ";
                System .out. println (" INPUT : " + input );
                System .out. println ("");
                String [] patt = {" engines . More engines "," ines
                \ nmore "," engines$ "};
                for (int i = 0; i < patt . length ; i ++)
                {
                        System .out. println (" PATTERN " + patt [i]);
                        boolean found ;
                        Pattern p1l = Pattern . compile ( patt [i]);
found = p1l. matcher ( input ). find ();
System .out. println (" DEFAULT match " + found );
Pattern pml =
                                Pattern . compile ( patt [i], Pattern .
                                DOTALL | Pattern . MULTILINE );
found = pml. matcher ( input ). find ();
System .out. println (" MultiLine match " + found );
System .out. println ();
}}}
```

Output:

```
INPUT : I dream of engines
more engines , all day long
PATTERN engines . more engines
DEFAULT match false
MultiLine match true
PATTERN ines
more
DEFAULT match true
MultiLine match true
PATTERN engines$
DEFAULT match false
MultiLine match true
```

End of Chapter