Java Regular Expressions

1. Java Regex - Java Regular Expressions (java.util.regex)
2. Java Regex - Pattern (java.util.regex.Pattern)
3. Java Regex - Matcher (java.util.regex.Matcher)
4. **Java Regex - Regular Expression Syntax**

Java Regex - Regular Expression Syntax

Jakob Jenkov
Last update: 2014-12-17

To use regular expressions effectively in Java, you need to know the syntax. The syntax is extensive, enabling you to write very advanced regular expressions. It may take a lot of exercise to fully master the syntax.

In this text I will go through the basics of the syntax with examples. I will not cover every little detail of the syntax, but focus on the main concepts you need to understand, in order to work with regular expressions. For a full explanation, see the **`Pattern` class JavaDoc page**.

Basic Syntax

Before showing all the advanced options you can use in Java regular expressions, I will give you a quick run-down of the Java regular expression syntax basics.

Characters

The most basic form of regular expressions is an expression that simply matches certain characters. Here is an example:

```
John
```

This simple regular expression will match occurences of the text "John" in a given input text.

You can use any characters in the alphabet in a regular expression.

You can also refer to characters via their octal, hexadecimal or unicode codes. Here are two examples:

```
\0101
\x41
\u0041
```

hexadecimal code (41) and the third uses the unicode code (0041).

Character Classes

Character classes are constructst that enable you to specify a match against multiple characters instead of just one. In other words, a character class matches a single character in the input text against multiple allowed characters in the character class. For instance, you can match either of the characters `a`, `b` or `c` like this:

```
[abc]
```

Character classes are nested inside a pair of square brackets `[]`. The brackets themselves are not part of what is being matched.

You can use character classes for many things. For instance, this example finds all occurrences of the word `John`, with either a lowercase or uppercase `J`:

```
[Jj]ohn
```

The character class `[Jj]` will match either a `J` or a `j`, and the rest of the expression will match the characters `ohn` in that exact sequence.

There are several other character classes you can use. See the character class table later in this text.

Predefined Character Classes

The Java regular expression syntax has a few predefined character classes you can use. For instance, the `\d` character class matches any digit, the `\s` character class matches any white space character, and the `\w` character matches any word character.

The predefined character classes do not have to be enclosed in square brackets, but you can if you want to combine them. Here are a few examples:

```
\d
[\d\s]
```

The first example matches any digit character. The second example matches any digit or any white space character.

The predefined character classes are listed in a table later in this text.

The syntax also include matchers for matching boundaries, like boundaries between words, the beginning and end of the input text etc. For instance, the `\w` matches boundaries between words, the `^` matches the beginning of a line, and the `$` matches the end of a line.

Here is a boundary matcher example:

```
^This is a single line$
```

This expression matches a line of text with only the text `This is a single line`. Notice the start-of-line and end-of-line matchers in the expression. These state that there can be nothing before or after the text, except the beginning and end of a line.

There is a full list of boundary matchers later in this text.

Quantifiers

Quantifiers enables you to match a given expression or subexpression multiple times. For instance, the following expression matches the letter `A` zero or more times:

```
A*
```

The `*` character is a quantifier that means "zero or more times". There is also a `+` quantifier meaning "one or more times", a `?` quantifier meaning "zero or one time", and a few others which you can see in the quantifier table later in this text.

Quantifiers can be either "reluctant", "greedy" or "possesive". A reluctant quantifier will match as little as possible of the input text. A greedy quantifier will match as much as possible of the input text. A possesive quantifier will match as much as possible, even if it makes the rest of the expression not match anything, and the expression to fail finding a match.

I will illustrate the difference between reluctant, greedy and possesive quantifiers with an example. Here is an input text:

```
John went for a walk, and John fell down, and John hurt his knee.
```

Then look at the following expression with a reluctant quantifier:

```
John.*?
```

Being a reluctant quantifier, the quantifier will match as little as possible, meaning zero characters. The expression will thus find the word `John` with zero characters after, 3 times in the above input text.

If we change the quantifier to a greedy quantifier, the expression will look like this:


```
John.*
```

The greedy quantifier will match as many characters as possible. Now the expression will only match the first occurrence of `John`, and the greedy quantifier will match the rest of the characters in the input text. Thus, only a single match is found.

Finally, lets us change the expression a bit to contain a possesive quantifier:

```
John.*+hurt
```

The `+` after the `*` makes it a possesive quantifier.

This expression will not match the input text given above, even if both the words `John` and `hurt` are found in the input text. Why is that? Because the `.*+` is possesive. Instead of matching as much as possible to make the expression match, as a greedy quantifier would have done, the possesive quantifier matches as much as possible, regardless of whether the expression will match or not.

The `.*+` will match all characters after the first occurrence of `John` in the input text, including the word `hurt`. Thus, there is no `hurt` word left to match, when the possesive quantifier has claimed its match.

If you change the quantifier to a greedy quantifier, the expression will match the input text one time. Here is how the expression looks with a greedy quantifier:

```
John.*hurt
```

You will have to play around with the different quantifiers and types to understand how they work. See the table later in this text for a full list of quantifiers.

Logical Operators

The Java regular expression syntax also has support for a few logical operators (and, or, not).

word `hurt`.

Characters

Construct	Matches
`x`	The character x. Any character in the alphabet can be used in place of x.
`\\`	The backslash character. A single backslash is used as escape character in conjunction with other characters to signal special matching, so to match just the backslash character itself, you need to escape with a backslash character. Hence the double backslash to match a single backslash character.
`\0n`	The character with octal value `0n`. n has to be between 0 and 7.
`\0nn`	The character with octal value `0nn`. n has to be between 0 and 7.
`\0mnn`	The character with octal value `0mnn`. m has to be between 0 and 3, n has to be between 0 and 7.
`\xhh`	The character with the hexadecimal value `0xhh`.
`\uhhhh`	The character with the hexadecimal value `0xhhhh`. This construct is used to match unicode characters.
`\t`	The tab character.
`\n`	The newline (line feed) character (unicode: `'\u000A'`).
`\r`	The carriage-return character (unicode: `'\u000D'`).
`\f`	The form-feed character (unicode: `'\u000C'`).
`\a`	The alert (bell) character (unicode: `'\u0007'`).
`\e`	The escape character (unicode: `'\u001B'`).
`\cx`	The control character corresponding to `x`

Character Classes

Construct	Matches
`[abc]`	Matches `a`, or `b` or `c`. This is called a simple class, and it matches any of the characters in the class.
`[^abc]`	Matches any character except `a`, `b`, and `c`. This is a negation.
`[a-zA-Z]`	Matches any character from `a` to `z`, or `A` to `Z`, including `a`, `A`, `z` and `Z`. This called a range.
`[a-d[m-p]]`	Matches any character from `a` to `d`, or from `m` to `p`. This is called a union.
`[a-z&&[def]]`	Matches `d`, `e`, or `f`. This is called an intersection (here between the range `a-z` and the characters `def`).
`[a-z&&[^bc]]`	Matches all characters from `a` to `z` except `b` and `c`. This is called a subtraction.
`[a-z&&[^m-p]]`	Matches all characters from `a` to `z` except the characters from `m` to `p`. This is also called a subtraction.

Construct	Matches
.	Matches any single character. May or may not match line terminators, depending on what flags were used to compile the `Pattern`.
\d	Matches any digit [0-9]
\D	Matches any non-digit character [^0-9]
\s	Matches any white space character (space, tab, line break, carriage return)
\S	Matches any non-white space character.
\w	Matches any word character.
\W	Matches any non-word character.

Boundary Matchers

Construct	Matches
^	Matches the beginning of a line.
$	Matches then end of a line.
\b	Matches a word boundary.
\B	Matches a non-word boundary.
\A	Matches the beginning of the input text.
\G	Matches the end of the previous match
\Z	Matches the end of the input text except the final terminator if any.
\z	Matches the end of the input text.

Quantifiers

Greedy	Reluctant	Possessive	Matches
X?	X??	X?+	Matches X once, or not at all (0 or 1 time).
X*	X*?	X*+	Matches X zero or more times.
X+	X+?	X++	Matches X one or more times.
X{n}	X{n}?	X{n}+	Matches X exactly n times.
X{n,}	X{n,}?	X{n,}+	Matches X at least n times.
X{n, m)	X{n, m)?	X{n, m)+	Matches X at least n time, but at most m times.

Logical Operators

X|Y	Matches X or Y.

Jakob Jenkov