The Java™ Tutorials

Trail: Essential Classes
Lesson: Regular Expressions

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

Quantifiers

Quantifiers allow you to specify the number of occurrences to match against. For convenience, the three sections of the Pattern API specification describing greedy, reluctant, and possessive quantifiers are presented below. At first glance it may appear that the quantifiers `X?`, `X??` and `X?+` do exactly the same thing, since they all promise to match "`X`, once or not at all". There are subtle implementation differences which will be explained near the end of this section.

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

Let's start our look at greedy quantifiers by creating three different regular expressions: the letter "a" followed by either `?`, `*`, or `+`. Let's see what happens when these expressions are tested against an empty input string `""`:

```
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.
```

Zero-Length Matches

In the above example, the match is successful in the first two cases because the expressions `a?` and `a*` both allow for zero occurrences of the letter `a`. You'll also notice that the start and end indices are both zero, which is unlike any of the examples we've seen so far. The empty input string `""` has no length, so the test simply matches nothing at index 0. Matches of this sort are known as a *zero-length matches*. A zero-length match can occur in several cases: in an empty input string, at the beginning of an input string, after the last character of an input string, or in between any two characters of an input string. Zero-length matches are easily identifiable because they always start and end at the same index position.

Let's explore zero-length matches with a few more examples. Change the input string to a single letter "a" and you'll notice something interesting:

```
Enter your regex: a?
Enter input string to search: a
I found the text "a" starting at index 0 and ending at index 1.
I found the text "" starting at index 1 and ending at index 1.

Enter your regex: a*
Enter input string to search: a
I found the text "a" starting at index 0 and ending at index 1.
I found the text "" starting at index 1 and ending at index 1.

Enter your regex: a+
```

```
Enter input string to search: a
I found the text "a" starting at index 0 and ending at index 1.
```

All three quantifiers found the letter "a", but the first two also found a zero-length match at index 1; that is, after the last character of the input string. Remember, the matcher sees the character "a" as sitting in the cell between index 0 and index 1, and our test harness loops until it can no longer find a match. Depending on the quantifier used, the presence of "nothing" at the index after the last character may or may not trigger a match.

Now change the input string to the letter "a" five times in a row and you'll get the following:

```
Enter your regex: a?
Enter input string to search: aaaaa
I found the text "a" starting at index 0 and ending at index 1.
I found the text "a" starting at index 1 and ending at index 2.
I found the text "a" starting at index 2 and ending at index 3.
I found the text "a" starting at index 3 and ending at index 4.
I found the text "a" starting at index 4 and ending at index 5.
I found the text "" starting at index 5 and ending at index 5.

Enter your regex: a*
Enter input string to search: aaaaa
I found the text "aaaaa" starting at index 0 and ending at index 5.
I found the text "" starting at index 5 and ending at index 5.

Enter your regex: a+
Enter input string to search: aaaaa
I found the text "aaaaa" starting at index 0 and ending at index 5.
```

The expression `a?` finds an individual match for each character, since it matches when "a" appears zero or one times. The expression `a*` finds two separate matches: all of the letter "a"'s in the first match, then the zero-length match after the last character at index 5. And finally, `a+` matches all occurrences of the letter "a", ignoring the presence of "nothing" at the last index.

At this point, you might be wondering what the results would be if the first two quantifiers encounter a letter other than "a". For example, what happens if it encounters the letter "b", as in "ababaaaab"?

Let's find out:

```
Enter your regex: a?
Enter input string to search: ababaaaab
I found the text "a" starting at index 0 and ending at index 1.
I found the text "" starting at index 1 and ending at index 1.
I found the text "a" starting at index 2 and ending at index 3.
I found the text "" starting at index 3 and ending at index 3.
I found the text "a" starting at index 4 and ending at index 5.
I found the text "a" starting at index 5 and ending at index 6.
I found the text "a" starting at index 6 and ending at index 7.
I found the text "a" starting at index 7 and ending at index 8.
I found the text "" starting at index 8 and ending at index 8.
I found the text "" starting at index 9 and ending at index 9.

Enter your regex: a*
Enter input string to search: ababaaaab
I found the text "a" starting at index 0 and ending at index 1.
I found the text "" starting at index 1 and ending at index 1.
I found the text "a" starting at index 2 and ending at index 3.
I found the text "" starting at index 3 and ending at index 3.
I found the text "aaaa" starting at index 4 and ending at index 8.
I found the text "" starting at index 8 and ending at index 8.
I found the text "" starting at index 9 and ending at index 9.

Enter your regex: a+
Enter input string to search: ababaaaab
I found the text "a" starting at index 0 and ending at index 1.
I found the text "a" starting at index 2 and ending at index 3.
I found the text "aaaa" starting at index 4 and ending at index 8.
```

Even though the letter "b" appears in cells 1, 3, and 8, the output reports a zero-length match at those locations. The regular expression `a?` is not specifically looking for the letter "b"; it's merely looking for the presence (or lack thereof) of the letter "a". If the quantifier allows for a match of "a" zero times, anything in the input string that's not an "a" will show up as a zero-length match. The remaining a's are matched according to the rules discussed in the previous examples.

To match a pattern exactly *n* number of times, simply specify the number inside a set of braces:

```
Enter your regex: a{3}
Enter input string to search: aa
No match found.

Enter your regex: a{3}
Enter input string to search: aaa
I found the text "aaa" starting at index 0 and ending at index 3.

Enter your regex: a{3}
Enter input string to search: aaaa
I found the text "aaa" starting at index 0 and ending at index 3.
```

Here, the regular expression `a{3}` is searching for three occurrences of the letter "a" in a row. The first test fails because the input string does not have enough a's to match against. The second test contains exactly 3 a's in the input string, which triggers a match. The third test also triggers a match because there are exactly 3 a's at the beginning of the input string. Anything following that is irrelevant to the first match. If the pattern should appear again after that point, it would trigger subsequent matches:

```
Enter your regex: a{3}
Enter input string to search: aaaaaaaaa
I found the text "aaa" starting at index 0 and ending at index 3.
I found the text "aaa" starting at index 3 and ending at index 6.
I found the text "aaa" starting at index 6 and ending at index 9.
```

To require a pattern to appear at least *n* times, add a comma after the number:

```
Enter your regex: a{3,}
Enter input string to search: aaaaaaaaa
I found the text "aaaaaaaaa" starting at index 0 and ending at index 9.
```

With the same input string, this test finds only one match, because the 9 a's in a row satisfy the need for "at least" 3 a's.

Finally, to specify an upper limit on the number of occurances, add a second number inside the braces:

```
Enter your regex: a{3,6} // find at least 3 (but no more than 6) a's in a row
Enter input string to search: aaaaaaaaa
I found the text "aaaaaa" starting at index 0 and ending at index 6.
I found the text "aaa" starting at index 6 and ending at index 9.
```

Here the first match is forced to stop at the upper limit of 6 characters. The second match includes whatever is left over, which happens to be three a's — the mimimum number of characters allowed for this match. If the input string were one character shorter, there would not be a second match since only two a's would remain.

Capturing Groups and Character Classes with Quantifiers

Until now, we've only tested quantifiers on input strings containing one character. In fact, quantifiers can only attach to one character at a time, so the regular expression "abc+" would mean "a, followed by b, followed by c one or more times". It would not mean "abc" one or more times. However, quantifiers can also attach to Character Classes and Capturing Groups, such as `[abc]+` (a or b or c, one or more times) or `(abc)+` (the group "abc", one or more times).

Let's illustrate by specifying the group `(dog)`, three times in a row.

```
Enter your regex: (dog){3}
Enter input string to search: dogdogdogdogdogdog
I found the text "dogdogdog" starting at index 0 and ending at index 9.
I found the text "dogdogdog" starting at index 9 and ending at index 18.

Enter your regex: dog{3}
Enter input string to search: dogdogdogdogdogdog
No match found.
```

Here the first example finds three matches, since the quantifier applies to the entire capturing group. Remove the parentheses, however, and the match fails because the quantifier `{3}` now applies only to the letter "g".

Similarly, we can apply a quantifier to an entire character class:

```
Enter your regex: [abc]{3}
Enter input string to search: abccabaaaccbbbc
I found the text "abc" starting at index 0 and ending at index 3.
I found the text "cab" starting at index 3 and ending at index 6.
I found the text "aaa" starting at index 6 and ending at index 9.
I found the text "ccb" starting at index 9 and ending at index 12.
```

```
I found the text "bbc" starting at index 12 and ending at index 15.

Enter your regex: abc{3}
Enter input string to search: abccabaaaccbbbc
No match found.
```

Here the quantifier `{3}` applies to the entire character class in the first example, but only to the letter "c" in the second.

Differences Among Greedy, Reluctant, and Possessive Quantifiers

There are subtle 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.

To illustrate, consider the input string `xfooxxxxxxfoo`.

```
Enter your regex: .*foo  // greedy quantifier
Enter input string to search: xfooxxxxxxfoo
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: xfooxxxxxxfoo
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: xfooxxxxxxfoo
No match found.
```

The first example uses the greedy quantifier `.*` to find "anything", zero or more times, followed by the letters `"f" "o" "o"`. Because the quantifier is greedy, the `.*` portion of the expression first eats the entire input string. At this point, the overall expression cannot succeed, because the last three letters (`"f" "o" "o"`) have already been consumed. So the matcher slowly backs off one letter at a time until the rightmost occurrence of "foo" has been regurgitated, at which point the match succeeds and the search ends.

The second example, however, is reluctant, so it starts by first consuming "nothing". Because "foo" doesn't appear at the beginning of the string, it's forced to swallow the first letter (an "x"), which triggers the first match at 0 and 4. Our test harness continues the process until the input string is exhausted. It finds another match at 4 and 13.

The third example fails to find a match because the quantifier is possessive. In this case, the entire input string is consumed by `.*+`, leaving nothing left over to satisfy the "foo" at the end of the expression. Use a possessive quantifier for situations where you want to seize all of something without ever backing off; it will outperform the equivalent greedy quantifier in cases where the match is not immediately found.

Previous page: Predefined Character Classes
Next page: Capturing Groups