Java Reference - Language - Regular Expressions

Table Of Contents

Selected Examples

Matches

The following examples where generated using (Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java

Literal

```
// Code
final static String regex = "foo";
final static String stringToSearch = "foo";

// Output
regex: foo
stringToSearch: foo
Match (1) of text "foo" starting at index 0 and ending at 3.
```

Single Character (Tab)

```
// Code
final static String regex = "foo\t";
final static String stringToSearch = "foo   bar";

// Output
regex: foo
stringToSearch: foo  bar
Match (1) of text "foo       " starting at index 0 and ending at 4.
```

Listed character set

```
// Include
regex: [fd]oo
stringToSearch: foo coo doo
Match (1) of text "foo" starting at index 0 and ending at 3.
Match (2) of text "doo" starting at index 8 and ending at 11.

// Exclude
regex: [^fd]oo
stringToSearch: foo coo doo
Match (1) of text "coo" starting at index 4 and ending at 7.
```

Ranged character set

```
// Include
regex: [a-z]oo
stringToSearch: foo Foo
Match (1) of text "foo" starting at index 0 and ending at 3.

// Exclude
regex: [^a-z]oo
stringToSearch: foo Foo
Match (1) of text "Foo" starting at index 4 and ending at 7.
```

Combined character sets (subtraction)

```
regex: [a-zA-Z&&[^cD]]oo
stringToSearch: foo Foo coo Doo
```

```
Match (1) of text "foo" starting at index 0 and ending at 3.
Match (2) of text "Foo" starting at index 4 and ending at 7.
```

Character Set, predefined (digit)

```
// Include
final static String regex = "\\d";

regex: \d
stringToSearch: H2O
Match (1) of text "2" starting at index 1 and ending at 2.

// Exclude
final static String regex = "\\D";

regex: \D
stringToSearch: H2O
Match (1) of text "H" starting at index 0 and ending at 1.
Match (2) of text "O" starting at index 2 and ending at 3.
```

Character Set, POSIX

```
// Code
final static String regex = "\\p{Alnum}";

// Output
regex: \p{Alnum}
stringToSearch: H99#
Match (1) of text "H" starting at index 0 and ending at 1.
Match (2) of text "9" starting at index 1 and ending at 2.
Match (3) of text "9" starting at index 2 and ending at 3.
```

Character Set, java.lang.Character (Mirrored)

```
final static String regex = "\\p{javaMirrored}";

regex: \p{javaMirrored}
stringToSearch: LOL{ [he
Match (1) of text "{" starting at index 3 and ending at 4.
Match (2) of text "[" starting at index 5 and ending at 6.
```

Character Set, Unicode block.

```
// Code
final static String regex = "\\p{InHiragana}";

// Output
regex: \p{InHiragana}
stringToSearch: Come to ぇthe partyβ
Match (1) of text "ぇ" starting at index 8 and ending at 9.
```

Character Set, Unicode category constant.

```
// Code
final static String regex = "\\p{Sc}";  // Currency Symbol

// Output
regex: \p{Sc}
stringToSearch: Come - $Niceds+ ☺ ,to ぇthe partyβ
Match (1) of text "$" starting at index 7 and ending at 8.
```

Unicode, single character. Literal and hexadecimally referenced.

```
final static String regex = "\\u03B2";
final static String stringToSearch = "β Greek \u03B2";

regex: \u03B2
stringToSearch: β Greek β
Match (1) of text "β" starting at index 0 and ending at 1.
Match (2) of text "β" starting at index 8 and ending at 9.
replacement: @
replacementResult: @ Greek @
```

Logical Operations

```
// Logical Or
// Listed character set example for reference.
regex: [fo|bo]o
stringToSearch: foo boo
Match (1) of text "fo" starting at index 0 and ending at 2.
Match (2) of text "bo" starting at index 4 and ending at 6.

// Logical Or example by comparison
regex: (fo|bo)o
stringToSearch: foo boo
Match (1) of text "foo" starting at index 0 and ending at 3.
Match (2) of text "boo" starting at index 4 and ending at 7.
```

Quantifiers (greedy)

```
// Many times or, failing that, zero
regex: a*
stringToSearch: aa
Match (1) of text "aa" starting at index 0 and ending at 2.
Match (2) of text "" starting at index 2 and ending at 2.

// Once or, failing that, zero
regex: a?
stringToSearch: aa
Match (1) of text "a" starting at index 0 and ending at 1.
Match (2) of text "a" starting at index 1 and ending at 2.
Match (3) of text "" starting at index 2 and ending at 2.

// Many times or, failing that, once
regex: a+
stringToSearch: aa
Match (1) of text "aa" starting at index 0 and ending at 2.
```

Boundary Matches

```
// Word boundary
regex: dog\b
stringToSearch: dogcat dog dog
Match (1) of text "dog" starting at index 7 and ending at 10.
Match (2) of text "dog" starting at index 11 and ending at 14.

// Not a word boundary
regex: dog\B
stringToSearch: dogcat dog dog
Match (1) of text "dog" starting at index 0 and ending at 3.
```

Flag setting in code.

```
final static String regex = "dog";
final static String stringToSearch = "Dog dog";
```

```
final static String replacement = "@";

// Bitwise "or" to set both flags.
final static int flags = Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE;

Pattern pattern = Pattern.compile(BasicRegex.regex, flags);
Matcher matcher = pattern.matcher(stringToSearch);

while (matcher.find()) { ...
```

(Oracle, 2012) http://docs.oracle.com/javase/tutorial/essential/regex/pattern.html

Flag setting in the pattern.

```
final static String regex = "(?iux)dog# comments";
final static String stringToSearch = "Dog dog";
final static String replacement = "@";

// Substitute the matches (java.lang.String)
System.out.printf("Replace all \"%s\" matches of \"%s\" with \"%s\": %s%n", regex,
    stringToSearch, replacement, stringToSearch.replaceAll(regex, replacement));

// Output
Replace all "(?iux)dog# comments" matches of "Dog dog" with "@": @ @
```

(Oracle, 2012) http://docs.oracle.com/javase/tutorial/essential/regex/pattern.html

Flag setting in the pattern overrides flag setting in code.

```
final static String regex = "(?-m)^\\w{3}+";
final static String stringToSearch = "cat pussy T \r\n"
+ "dog ratpig";
final static String replacement = "@";

final static int flags = Pattern.MULTILINE;

...
Pattern pattern = Pattern.compile(BasicRegex.regex, flags);

// The "turn off multiline" in the pattern overrides the Pattern.MULTILINE setting in code.
regex: (?-m)^\w{3}+
stringToSearch: cat pussy T
dog ratpig
Match (1) of text "cat" starting at index 0 and ending at 3.
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.Java

Captures, General

```
regex: (\w{3})(dog)
stringToSearch: catdogpussyratdog
Match (1) of text "catdog" starting at index 0 and ending at 6.
  Capture groups: 2
  Capture group (0) of text "catdog" starting at index 0 and ending at 6.
  Capture group (1) of text "cat" starting at index 0 and ending at 3.
  Capture group (2) of text "dog" starting at index 3 and ending at 6.
Match (2) of text "ratdog" starting at index 11 and ending at 17.
  Capture groups: 2
  Capture group (0) of text "ratdog" starting at index 11 and ending at 17.
  Capture group (1) of text "rat" starting at index 11 and ending at 14.
  Capture group (2) of text "dog" starting at index 14 and ending at 17.
replacement: $2$1
replacementResult: dogcatpussydograt
```

Captures, Look ahead and Look behind

```
// Look ahead positive.
regex: (apple|cherry)(?= chocolate)
stringToSearch: Today's specials are apple chocolate pie and cherry banana pie.
Match (1) of text "apple" starting at index 21 and ending at 26.

// Look ahead negative.
regex: (apple|cherry)(?! chocolate)
stringToSearch: Today's specials are apple chocolate pie and cherry banana pie.
Match (1) of text "cherry" starting at index 45 and ending at 51.


// Look behind positive.
regex: (?<=fried )(bananas|clam)
stringToSearch: Tomorrow's special is fried bananas with baked clam.
Match (1) of text "bananas" starting at index 28 and ending at 35.

// Look behind negative
regex: (?<!fried )(bananas|clam)
stringToSearch: Tomorrow's special is fried bananas with baked clam.
Match (1) of text "clam" starting at index 47 and ending at 51.
```

Captures, backreference.

```
regex: (\w{3})(\1)
stringToSearch: catdogcatcat
Match (1) of text "catcat" starting at index 6 and ending at 12.
  Capture groups: 2
  Capture group (0) of text "catcat" starting at index 6 and ending at 12.
  Capture group (1) of text "cat" starting at index 6 and ending at 9.
  Capture group (2) of text "cat" starting at index 9 and ending at 12.
```

Replacement

```
regex: ([cr]at)(.at)
stringToSearch: catdogratbat
Match (1) of text "ratbat" starting at index 6 and ending at 12.
  Capture groups: 2
  Capture group (0) of text "ratbat" starting at index 6 and ending at 12.
  Capture group (1) of text "rat" starting at index 6 and ending at 9.
  Capture group (2) of text "bat" starting at index 9 and ending at 12.
replacement: #$2|$1|$0#
replacementResult: catdog#bat|rat|ratbat#
```

Conceptual Overview

Regular expressions are a way to describe a set of strings. It allows you to pattern match strings and perform replacements. The relevant parts involved in regular expression operations are the: pattern (or "regex"); flags; string-to-search (or the "input"); match(es); capture(s); replacement string; and replacement result.

```
// The regular expression (or "regex" or "pattern")
"(\w{3})(dog)# Comment"

// final static int flags = Pattern.CASE_INSENSITIVE | Pattern.COMMENTS;

// String to search (or "input string")
"catdogpussyratdog"

// Matches and captures
Match (1) of text "catdog" starting at index 0 and ending at 6.
  Capture groups: 2
  Capture group (0) of text "catdog" starting at index 0 and ending at 6.
  Capture group (1) of text "cat" starting at index 0 and ending at 3.
  Capture group (2) of text "dog" starting at index 3 and ending at 6.
Match (2) of text "ratdog" starting at index 11 and ending at 17.
  Capture groups: 2
  Capture group (0) of text "ratdog" starting at index 11 and ending at 17.
  Capture group (1) of text "rat" starting at index 11 and ending at 14.
  Capture group (2) of text "dog" starting at index 14 and ending at 17.

// Replacement String
"@$2cool$1@"

// Replacement Result
@dogcoolcat@pussy@dogcoolrat@
```

The term "Regular expression" and its abbreviation "regex" are ambiguous. Sometimes these refer to the whole string description and replacement apparatus. At other times they refer specifically to the pattern that describes the set of strings, the small (but key) component in the apparatus.

With the whole apparatus of regexes you generally want to do one of three things:

- Test whether a match has occurred against all or part of the string to search.
- Return all the matches and/or captures.
- Replace the captures (which might equal the matches).

Java Coding Operations

Overview

(Oracle, 2011) http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html

Regex operations are supported primarily by `java.util.regex` and secondarily by `java.lang.String`.

`java.util.regex` has three main classes of relevance: `Pattern`, `Matcher`, and `PatternSyntaxException`.

Follow those links to the respective SE7 documentation.

Paradigmatic Example

Paradigmatic use of the `java.util.regex` classes for matching, and replacement.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;

public class RegexDemo {
  public static void start() {
    BasicRegex. javaUtilRegexDemo();
  }

  private static class BasicRegex {
    final static String regex = "(\\w{3})(dog)# Comment";
    final static String stringToSearch = "catdogpussyratdog";
    final static String replacement = "@$2cool$1@";

    final static int flags = Pattern.CASE_INSENSITIVE | Pattern.COMMENTS;

    final static boolean displayCaptures = true;

    static String replacementResult = "";

    private static void javaUtilRegexDemo() {
      try {
        Pattern pattern = Pattern.compile(BasicRegex.regex, flags);
        Matcher matcher = pattern.matcher(stringToSearch);

        System.out.printf("regex: %s%n", pattern.pattern());
        System.out.printf("stringToSearch: %s%n", stringToSearch);

        int i = 1;
        boolean found = false;

        // List matches
        while (matcher.find()) {
          System.out.printf(
              "Match (%d) of text \"%s\" starting at index %d and ending at %d.%n", i++,
              matcher.group(), matcher.start(), matcher.end());

          if (displayCaptures) {
            System.out.printf("\tCapture groups: %d%n", matcher.groupCount());

            // List capture groups within each match
```

```
            for (int j = 0; j < matcher.groupCount() + 1; j++) {
              System.out.printf("\tCapture group (%d) of text \"%s\" starting"
                  + " at index %d and ending at %d.%n", j, matcher.group(j),
                  matcher.start(j), matcher.end(j));
            } // for (int j
          } // if (displayCaptures

          found = true;
        } // while (matcher

        if (!found) {
          System.out.println("No match found.");
        } else if (replacement.length() > 0) {
          System.out.printf("replacement: %s%n", replacement);
          replacementResult = matcher.replaceAll(replacement);
          System.out.printf("replacementResult: %s%n", replacementResult);
        }

      } catch (PatternSyntaxException pse) {
        System.err.format("There is a problem" + " with the regular expression!%n");
        System.err.format("The pattern in question is: %s%n", pse.getPattern());
        System.err.format("The description is: %s%n", pse.getDescription());
        System.err.format("The message is: %s%n", pse.getMessage());
      }
    } // javaUtilRegexDemo()

  } // BasicRegex

} // RegexDemo

// Output
regex: (\w{3})(dog)
stringToSearch: catdogpussyratdog
Match (1) of text "catdog" starting at index 0 and ending at 6.
  Capture groups: 2
  Capture group (0) of text "catdog" starting at index 0 and ending at 6.
  Capture group (1) of text "cat" starting at index 0 and ending at 3.
  Capture group (2) of text "dog" starting at index 3 and ending at 6.
Match (2) of text "ratdog" starting at index 11 and ending at 17.
  Capture groups: 2
  Capture group (0) of text "ratdog" starting at index 11 and ending at 17.
  Capture group (1) of text "rat" starting at index 11 and ending at 14.
  Capture group (2) of text "dog" starting at index 14 and ending at 17.
replacement: @$2cool$1@
replacementResult: @dogcoolcat@pussy@dogcoolrat@
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java
based on (Oracle, 2012) http://docs.oracle.com/javase/tutorial/essential/regex/test_harness.html

Common Single Line Methods

Match – Boolean Test

java.lang.String	Description	java.util.regex
public boolean matches(String regex)	Tells whether or not **all of this string** matches the given regular expression.	Pattern.matches(*regex*, *stringToSearch*)
	Tells whether or not **part of this string** matches the given regular expression.	Pattern.compile(*regex*). matcher(*stringToSearch*).find(); Pattern.compile(regex) .matcher(stringToSearc h).lookingAt();

```
final static String regex = "(\\w{3})(dog)";
final static String stringToSearch = "catdogpussycatdog";
final static String replacement = "$2cool$1";
```

```
private static void matches() {

  // Boolean match
  // Does the regex match the whole stringToSearch?
  System.out.printf("\"%s\" matches the whole of \"%s\"?: %s%n", regex,
      stringToSearch, stringToSearch.matches(regex));
  // "(\w{3})(dog)" matches the whole of "catdogpussycatdog"?: false

  // Does the regex match the whole stringToSearch?
  System.out.printf("\"%s\" matches the whole of \"%s\"?: %s%n", regex,
      stringToSearch, Pattern.matches(regex, stringToSearch));
  // "(\w{3})(dog)" matches the whole of "catdogpussycatdog"?: false

  // Does the regex match part of the stringToSearch?
  System.out.printf("\"%s\" matches part of \"%s\"?: %s%n", regex, stringToSearch,
      Pattern.compile(regex).matcher(stringToSearch).find());
  // "(\w{3})(dog)" matches part of "catdogpussycatdog"?: true

} // matches()
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java & (Oracle, 2012)
http://docs.oracle.com/javase/tutorial/essential/regex/matcher.html

Match – Substring Extraction

Return the first matched substring.

```
private static String matchedSubstring(String stringToSearch, String patternString) {
  Pattern pattern = Pattern.compile(patternString);
  Matcher matcher = pattern.matcher(stringToSearch);

  if (matcher.find()) {
    return matcher.group();
  } else {
    return "";
  }
}

System.out.println(matchedSubstring("This is 34 good", "\\d+"));

// Output
34
```

Replacement

java.lang.String	Description	java.util.regex
public String replace(CharSequence target, CharSequence replacement)	Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence. The replacement proceeds from the beginning of the string to the end, for example, replacing "aa" with "b" in the string "aaa" will result in "ba" rather than "ab".	
public String replaceFirst(String regex, String replacement)	Replaces the first substring of this string that matches the given regular expression with the given replacement	Pattern.compile(regex).matcher(str).replaceFirst(repl)
public String replaceAll(String regex, String replacement)	Replaces each substring of this string that matches the given regular expression with the given replacement.	Pattern.compile(regex).matcher(str).replaceAll(repl)

```
final static String regex = "(\\w{3})(dog)";
final static String stringToSearch = "catdogpussycatdog";
```

```
final static String replacement = "$2cool$1";

private static void replacement() {

  // Substitute the matches (java.lang.String)
  System.out.printf("Replace all \"%s\" matches of \"%s\" with \"%s\": %s%n",
      regex, stringToSearch, replacement,
      stringToSearch.replaceAll(regex, replacement));
  // Replace all "(\w{3})(dog)" matches of "catdogpussycatdog" with "$2cool$1":
  // dogcoolcatpussydogcoolcat

  // Substitute the matches (java.util.regex)
  System.out.printf("Replace all \"%s\" matches of \"%s\" with \"%s\": %s%n",
      regex, stringToSearch, replacement,
      Pattern.compile(regex).matcher(stringToSearch).replaceAll(replacement));
  // Replace all "(\w{3})(dog)" matches of "catdogpussycatdog" with "$2cool$1":
  // dogcoolcatpussydogcoolcat

} // replacement
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java

Split

java.lang.String	Description	java.util.regex
public String[] split(String regex, int limit)	Splits this string around matches of the given regular expression.	Pattern.compile(*regex*). split(*str, n*)
public String[] split(String regex)	Splits this string around matches of the given regular expression. This method works the same as if you invoked the two-argument split method with the given expression and a limit argument of zero. Trailing empty strings are not included in the resulting array	Pattern.compile(*regex*). split(*str*)

```
private static void splits() {

  String regex = "\\s*@\\s*";
  String stringToSplit = "Apples @  Oranges@Bananas @ Pears";
  int maxElements = 3; // 0 for all elements.

  System.out.printf(
      "\"%s\" defined to split \"%s\", into a max of %d elements: %s%n", regex,
      stringToSplit, maxElements,
      Arrays.toString(stringToSplit.split(regex, maxElements)));
  // "\s*@\s*" defined to split "Apples @ Oranges@Bananas @ Pears",
  // into a max of 3 elements: [Apples, Oranges, Bananas @ Pears]

  System.out.printf(
      "\"%s\" defined to split \"%s\", into a max of %d elements: %s%n", regex,
      stringToSplit, maxElements,
      Arrays.toString(Pattern.compile(regex).split(stringToSplit, maxElements)));
  // "\s*@\s*" defined to split "Apples @ Oranges@Bananas @ Pears",
  // into a max of 3 elements: [Apples, Oranges, Bananas @ Pears]

}
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java based on (Oracle, 2012) http://docs.oracle.com/javase/tutorial/essential/regex/pattern.html & http://docs.oracle.com/javase/tutorial/essential/regex/matcher.html

Other Matcher Class Methods

Index methods in the Matcher class are available after a `matcher.find()` and hold information about matches and, for each match, groups of captures.

- `public int start()`: *Match index:* Returns the start index of the previous match.
- `public int end()`: *Match index:* Returns the offset after the last character matched.
- `public int start(int group)`: *Capture Index for a Match:* Returns the start index of the subsequence captured by the given group during the previous match operation.
- `public int end(int group)`: *Capture Index for a Match:* Returns the offset after the last character of the subsequence captured by the given group during the previous match operation.

```
final static String regex = "(\\w{3})(dog)";
final static String stringToSearch = "catdogpussycatdog";

final static int flags = Pattern.CASE_INSENSITIVE | Pattern.COMMENTS;

Pattern pattern = Pattern.compile(BasicRegex.regex, flags);
Matcher matcher = pattern.matcher(stringToSearch);

int i = 1;

// List matches
while (matcher.find()) {
  System.out.printf(
      "Match (%d) of text \"%s\" starting at index %d and ending at %d.%n", i++,
      matcher.group(), matcher.start(), matcher.end());

  if (displayCaptures) {
    System.out.printf("\tCapture groups: %d%n", matcher.groupCount());

    // List capture groups within each match
    for (int j = 0; j < matcher.groupCount() + 1; j++) {
      System.out.printf("\tCapture group (%d) of text \"%s\" starting"
          + " at index %d and ending at %d.%n", j, matcher.group(j),
          matcher.start(j), matcher.end(j));
    } // for (int j
  } // if (displayCaputeres
} // while (matcher

// Output
regex: (\w{3})(dog)
stringToSearch: catdogpussycatdog
Match (1) of text "catdog" starting at index 0 and ending at 6.
  Capture groups: 2
  Capture group (0) of text "catdog" starting at index 0 and ending at 6.
  Capture group (1) of text "cat" starting at index 0 and ending at 3.
  Capture group (2) of text "dog" starting at index 3 and ending at 6.
Match (2) of text "catdog" starting at index 11 and ending at 17.
  Capture groups: 2
  Capture group (0) of text "catdog" starting at index 11 and ending at 17.
  Capture group (1) of text "cat" starting at index 11 and ending at 14.
  Capture group (2) of text "dog" starting at index 14 and ending at 17.
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java & (Oracle, 2012)
http://docs.oracle.com/javase/tutorial/essential/regex/matcher.html

Regex Syntax

Pattern ("regex") Syntax

Single Characters

Construct	Matches
Single characters	
x	The character *x*
\\	The backslash character
\0*n*	The character with octal value 0*n* (0 <= *n* <= 7)
\0*nn*	The character with octal value 0*nn* (0 <= *n* <= 7)
\0*mnn*	The character with octal value 0*mnn* (0 <= *m* <= 3, 0 <= *n* <= 7)
\x*hh*	The character with hexadecimal value 0x*hh*
\u*hhhh*	The character with hexadecimal value 0x*hhhh*
\x{*h...h*}	The character with hexadecimal value 0x*h...h* (`Character.MIN_CODE_POINT` <= 0x*h...h* <= `Character.MAX_CODE_POINT`)
\t	The tab character ('\u0009')
\r	The carriage-return character ('\u000D')
\n	The newline (line feed) character ('\u000A')
\f	The form-feed character ('\u000C')
\a	The alert (bell) character ('\u0007')
\e	The escape character ('\u001B')
\c*x*	The control character corresponding to *x*

(Oracle, 2012) *http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html*

Character sets

Construct	Matches
Listed character sets	
[abc]	a, b, or c (included)
[^abc]	Any character except a, b, or c (excluded)
Ranged character sets.	
[a-z]	a through z (included)
[^a-z]	Any character except a through z (excluded)

Combined character sets.	
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)
Predefined character sets	
.	Any character (may or may not match `line terminators`)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character. That is, identifier legal: [a-zA-Z_0-9]
\W	A non-word character. That is, not identifier legal: [^\w]
POSIX character sets (US-ASCII only, don't use for handling unicode characters)	
\p{Lower}	A lower-case alphabetic character: [a-z]
\p{Upper}	An upper-case alphabetic character:[A-Z]
\p{ASCII}	All ASCII:[\x00-\x7F]
\p{Alpha}	An alphabetic character:[\p{Lower}\p{Upper}]
\p{Digit}	A decimal digit: [0-9]
\p{Alnum}	An alphanumeric character:[\p{Alpha}\p{Digit}]
\p{Punct}	Punctuation: One of !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
\p{Graph}	A visible character: [\p{Alnum}\p{Punct}]
\p{Print}	A printable character: [\p{Graph}\x20]
\p{Blank}	A space or a tab: [\t]
\p{Cntrl}	A control character: [\x00-\x1F\x7F]
\p{XDigit}	A hexadecimal digit: [0-9a-fA-F]
\p{Space}	A whitespace character: [\t\n\x0B\f\r]
java.lang.Character **sets (simple java character type). Unicode Safe.**	
\p{javaLetter}	Equivalent to java.lang.Character.isLetter() [E.g. "利Gre"]
\p{javaDigit}	Equivalent to java.lang.Character.isDigit() [E.g. "78٢", the last is Arabic 7"]
\p{javaLetterOrDigit}	Equivalent to java.lang.Character.isLetterOrDigit()
\p{javaLowerCase}	Equivalent to java.lang.Character.isLowerCase()

\p{javaUpperCase}	Equivalent to java.lang.Character.isUpperCase()
\p{javaWhitespace}	Equivalent to java.lang.Character.isWhitespace()
\p{javaMirrored}	Equivalent to java.lang.Character.isMirrored() [(){}[]<>]
Character sets for Unicode scripts, blocks, categories and binary properties	
\p{IsLatin}	A Latin script character (`script`) (SE7)
\p{IsAlphabetic}	An alphabetic character (`binary property`) (SE7)
\p{InGreek} \p{InBasic_Latin}	A character in the Greek block A character in the Basic Latin block (`block constants`)
\p{Lu} \p{Ll} \p{L} \p{Sm} \p{Sc} \p{S}	Letter, uppercase. Letter, lowercase. Letter, any. Symbol, maths. Symbol, currency. Symbol, any. (`category constants`)
\P{InGreek}	Any character except one in the Greek block (negation)
[\p{L}&&[^\p{Lu}]]	Any letter except an uppercase letter (subtraction)

(Oracle, 2012) http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

Logical Operations

XY	X followed by Y
X|Y	Either X or Y
(X)	X, as a capturing group

Quantifiers

Greedy	**Reluctant**	**Possessive**	**Meaning**
X*	X*?	X*+	X, many times or, failing that, zero
X?	X??	X?+	X, once or, failing that, zero
X+	X+?	X++	X, many times or, failing that, once
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

(Oracle, 2012) http://docs.oracle.com/javase/tutorial/essential/regex/quant.html

Boundary Matches

Construct	Matches
Boundary matchers	
^	The beginning of input. If multiline flag set: beginning of line.
$	The end of input. If multiline flag set: end of line.
\A	The beginning of the input
\Z	The end of the input but for the final terminator, if any
\z	The end of the input
\b	A word boundary
\B	A non-word boundary
\G	The end of the previous match

Flags

Constant	Equivalent Embedded Flag Expression	Description
Pattern.CASE_INSENSITIVE	(?i)	Enables case-insensitive matching. Case-insensitive matching assumes that only characters in the US-ASCII charset are being matched. Unicode-aware case-insensitive matching can be enabled by specifying the UNICODE_CASE flag in conjunction with this flag.
Pattern.UNICODE_CASE	(?u)	Enables Unicode-aware case folding. When this flag is specified then case-insensitive matching, when enabled by the CASE_INSENSITIVE flag, is done in a manner consistent with the Unicode Standard. By default, case-insensitive matching assumes that only characters in the US-ASCII charset are being matched.
Pattern.MULTILINE	(?m)	Enables multiline mode. In multiline mode the expressions `^` and `$` match operate with respect to each line (for strings-to-search separated by a line terminator). When flag not set: these expressions only match at the beginning and the end of the entire input sequence.
Pattern.DOTALL	(?s)	Enables dotall mode. In dotall mode, the expression `.` matches any character, including a line terminator. By default this expression does not match line terminators.
Pattern.UNIX_LINES	(?d)	Enables Unix lines mode. In this mode, only the `'\n'` line terminator is recognized in the behavior of `.`, `^`, and `$`.
Pattern.LITERAL	None	Enables literal parsing of the pattern. When this flag is specified then the pattern is treated as a sequence of literal characters. Metacharacters or escape sequences in the pattern will be given no special meaning. The flags CASE_INSENSITIVE and UNICODE_CASE retain their impact on matching when used in conjunction with this flag. The other flags become superfluous

Pattern.COMMENTS	(?x)	Permits whitespace and comments in the pattern. In this mode, whitespace is ignored, and embedded comments starting with # are ignored until the end of a line.
Pattern.CANON_EQ	None	Enables canonical equivalence. When this flag is specified, two characters will be considered to match if, and only if, their full canonical decompositions match. The expression "a\u030A" (°), for example, will match the string "\u030A" (°) when this flag is specified. By default, matching does not take canonical equivalence into account.

(Oracle, 2012) *http://docs.oracle.com/javase/tutorial/essential/regex/pattern.html*

Quotation

\	Nothing, but quotes the following character
\Q	Nothing, but quotes all characters until \E
\E	Nothing, but ends quoting started by \Q

(Oracle, 2012) *http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html*

Captures

Basic Capture	
(X)	X, as a capturing group
Back references	
\n	Whatever the nth capturing group matched. n = 1 to 9. n = 0 throws exception.
\k<name> (SE7)	Whatever the named-capturing group "name" matched
Special constructs (named-capturing)	
(?<name>X) (SE7)	X, as a named-capturing group
Special constructs (non-capturing)	
(?:X)	X, as a non-capturing group
(?idmsux-idmsux)	Nothing, but turns match flags i d m s u x on - off
(?idmsux-idmsux:X)	X, as a non-capturing group with the given flags i d m s u x on - off
(?=X)	X, via zero-width positive lookahead (atomic)
(?!X)	X, via zero-width negative lookahead (atomic)
(?<=X)	X, via zero-width positive lookbehind (atomic)
(?<!X)	X, via zero-width negative lookbehind (atomic)
(?>X)	X, as an independent, non-capturing group (atomic)

Replacement Syntax

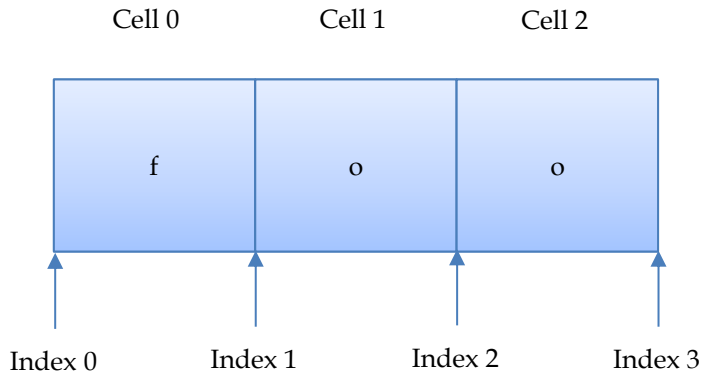
Capturing reference	
$g [g = '0' ... '9']	Capture group. '0' is the whole group. '1' is the first match.
${name} (SE 7)	Named capture group.
\$	Literal dollar sign.
\x	Escape literal characters

(Oracle, 2012)
http://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html#appendReplacement%28java.lang.StringBuffer,%20java.lang.String%29

Regex Rules

General

Match indices are counted such that characters occupy cells and indices point to the cell boundaries, starting at the left edge, at zero. The end index comes *after* the cell of the last character.




```
regex: foo
stringToSearch: foo
Match (1) of text "foo" starting at index 0 and ending at 3.
```

(Oracle, 2012) *http://docs.oracle.com/javase/tutorial/essential/regex/literals.html*

When using the backslash ('\') character in Java Strings, where Java does not recognize the escape sequence in string terms, you'll have to escape the backslash character with another one.

```
// Java String doesn't recognize "\.". Therefore escape the backslash.
final static String regex = "cat\\.";
final static String stringToSearch = "cat.";

// Regex Operation Output
regex: cat\.
stringToSearch: cat.
Match (1) of text "cat." starting at index 0 and ending at 4.


// Java String recognizes "\t". Therefore don't escape the backslash.
final static String regex = "cat\theat";
final static String stringToSearch = "cat   heat";

// Regex Operation Output
regex: cat    heat
stringToSearch: cat   heat
Match (1) of text "cat       heat" starting at index 0 and ending at 8.
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java

Regex pattern metacharacters: < ([{ \ ^ - = $! |] }) ? * + . >

(Oracle, 2012) *http://docs.oracle.com/javase/tutorial/essential/regex/literals.html*

Quantifiers

A meaning of "zero" in "... or, failing that, zero" times for quantification.

```
// Search for "foo" and any character (.), once or, failing that, zero.
final static String regex = "foo.?";
final static String stringToSearch = "fooxfoo";

regex: foo.?
stringToSearch: fooxfoo
Match (1) of text "foox" starting at index 0 and ending at 4.  // "Once" match
Match (2) of text "foo" starting at index 4 and ending at 7.  // "Zero" match
```

Zero length matches can occur with: an empty regex pattern; an empty string to search; between any two characters of an string to search; or at the beginning or end of an string to search

```
// Empty regex pattern.
final static String regex = "";
final static String stringToSearch = "cat";
regex:
stringToSearch: cat
Match (1) of text "" starting at index 0 and ending at 0.
Match (2) of text "" starting at index 1 and ending at 1.
Match (3) of text "" starting at index 2 and ending at 2.
Match (4) of text "" starting at index 3 and ending at 3.

// Empty string to search
final static String regex = "a*";
final static String stringToSearch = "";
regex: a*
stringToSearch:
Match (1) of text "" starting at index 0 and ending at 0.

// Between characters of string to search
final static String regex = "a*";
final static String stringToSearch = "cat";
regex: a*
stringToSearch: cat
Match (1) of text "" starting at index 0 and ending at 0.
Match (2) of text "a" starting at index 1 and ending at 2.
Match (3) of text "" starting at index 2 and ending at 2.
Match (4) of text "" starting at index 3 and ending at 3.

// At the end of the string to search
final static String regex = "a*";
final static String stringToSearch = "aaa";
regex: a*
stringToSearch: aaa
Match (1) of text "aaa" starting at index 0 and ending at 3.
Match (2) of text "" starting at index 3 and ending at 3.
```

(Oracle, 2012) http://docs.oracle.com/javase/tutorial/essential/regex/quant.html

Note

```
regex: a*
stringToSearch: aaa
Match (1) of text "aaa" starting at index 0 and ending at 3.
Match (2) of text "" starting at index 3 and ending at 3.

regex: aa*
stringToSearch: aaa
Match (1) of text "aaa" starting at index 0 and ending at 3.
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java

Differences between greedy quantifiers:

- (*) many times or, failing that, zero;
- (?) once or, failing that, zero; and
- (+) many times or, failing that, once.

```
// Many times or, failing that, zero
regex: a*
stringToSearch: aa
Match (1) of text "aa" starting at index 0 and ending at 2.
Match (2) of text "" starting at index 2 and ending at 2.

// Once or, failing that, zero
regex: a?
stringToSearch: aa
Match (1) of text "a" starting at index 0 and ending at 1.
Match (2) of text "a" starting at index 1 and ending at 2.
Match (3) of text "" starting at index 2 and ending at 2.

// Many times or, failing that, once
regex: a+
stringToSearch: aa
Match (1) of text "aa" starting at index 0 and ending at 2.
```

(Oracle, 2012) *http://docs.oracle.com/javase/tutorial/essential/regex/quant.html*

Quantifiers can attach to *Character Set Ranges* and *Capturing Groups* (as well as single characters and other character sets).

```
// Quantified Character Set Ranges
regex: [abc]{3}
stringToSearch: aaaabcabcabc
Match (1) of text "aaa" starting at index 0 and ending at 3.
Match (2) of text "abc" starting at index 3 and ending at 6.
Match (3) of text "abc" starting at index 6 and ending at 9.
Match (4) of text "abc" starting at index 9 and ending at 12.

// Quantified Capturing Group
regex: (abc){3}
stringToSearch: aaaabcabcabc
Match (1) of text "abcabcabc" starting at index 3 and ending at 12.

// Note the following regex (pattern) only quantifies the single character 'c'.
regex: abc{3}
stringToSearch: aaaabcabcabc
No match found.
```

(Oracle, 2012) *http://docs.oracle.com/javase/tutorial/essential/regex/quant.html*

The differences between greedy, reluctant and possessive quantifiers.

A regex pattern with a quantifier is processed against that part of the string-to-search (the 'input' string) that hasn't yet been matched in three steps:

- The quantified portion of the regex pattern eats the whole string-to-search in some manner:
    a. Greedy: Eats the whole string-to-search
    b. Reluctant (?): Doesn't eat any of the whole string-to-search.
    c. Possessive (+): Eats the whole string-to-search.
- The whole regex pattern, not just the quantified portion of the regex pattern, is then compared to the eaten whole string-to-search (so for a Reluctant (?) quantifier there is no comparison).
- If there is no match then the string-to-search is further processed by:

a. Greedy: "backing off" the rightmost character in the string-to-search. The quantified portion of the regex pattern and the non-quantified portion of the regex pattern is now compared to the string-to-search (all the characters from left to right less one). If there is a match the process stops. Otherwise this is repeated until there are no more characters in the string-to-search.

b. Reluctant (?): Looking at the leftmost character in the string-to-search. The quantified portion of the regex pattern and the non-quantified portion of the regex pattern is now compared to the string-to-search (only the left most character). If there is a match this is noted. Match or not the process is repeated by appending the next leftmost character from the string-to-search until the string-to-search is exhausted.

c. Possessive (+): There is one, and only one, attempt at a match against the whole string-to-search. If there is no match the process stops (there is no backing off).

```
// Greedy [No symbol]
regex: .*foo
stringToSearch: xfooxxxfoo
Match (1) of text "xfooxxxfoo" starting at index 0 and ending at 10.

// Greedy Steps
// Quantified + Unquantified Pattern Resolution V stringToSearch.
"xfooxxxfoo" + "foo" V "xfooxxxfoo" -> No Match.
"xfooxxxfo" + "foo" V "xfooxxxfoo"  -> No Match.
"xfooxxxf" + "foo" V "xfooxxxfoo" -> No Match.
"xfooxxx" + "foo" V "xfooxxxfoo" -> Match ("xfooxxxfoo")


// Reluctant [?]
regex: .*?foo
stringToSearch: xfooxxxfoo
Match (1) of text "xfoo" starting at index 0 and ending at 4.
Match (2) of text "xxxfoo" starting at index 4 and ending at 10.

// Reluctant Steps
// Quantified + Unquantified Pattern Resolution V stringToSearch.
"" + "foo" V "xfooxxxfoo" -> No match
"x" + "foo" V "xfooxxxfoo" -> Match (1) of text "xfoo" starting at index 0 and ending at 4.
"xfoo" from "xfooxxxfoo" consumed
"x" + "foo" V "xxxfoo" -> No match
"xx" + "foo" V "xxxfoo" -> No match
"xxx" + "foo" v "xxxfoo" -> Match (2) of text "xxxfoo" starting at index 4 and ending at 10.
"xxxfoo" from "xxxfoo" consumed.


// Possessive [+]
regex: .*+foo
stringToSearch: xfooxxxfoo
No match found.

// Possessive steps
// Quantified + Unquantified Pattern Resolution V stringToSearch.
"xfooxxxfoo" + "foo" V "xfooxxxfoo" -> No Match.
```

(Oracle, 2012) http://docs.oracle.com/javase/tutorial/essential/regex/quant.html
(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java

Another example showing the differences between greedy, reluctant and possessive quantifiers.

```
// Greedy [No Symbol]
regex: .*foo
stringToSearch: xfooxxxfooa
Match (1) of text "xfooxxxfoo" starting at index 0 and ending at 10.

// Greedy Steps
// Quantified + Unquantified Pattern Resolution V stringToSearch.
"xfooxxxfooa" + "foo" V "xfooxxxfooa" -> No Match
"xfooxxxfoo" + "foo" V "xfooxxxfooa" -> No Match
"xfooxxxfo" + "foo" V "xfooxxxfooa" -> No Match
"xfooxxxf" + "foo" V "xfooxxxfooa" -> No Match
```

```
"xfooxxx" + "foo" V "xfooxxxfooa" -> Match (1) of text "xfooxxxfoo" starting at index 0 and
ending at 10.
"xfooxxxfoo" in "xfooxxxfooa" consumed leaving "a"
"a" + "foo" V "a" -> No Match
"" + "foo" V "a" -> No Match


// Reluctant [?]
regex: .*?foo
stringToSearch: xfooxxxfooa
Match (1) of text "xfoo" starting at index 0 and ending at 4.
Match (2) of text "xxxfoo" starting at index 4 and ending at 10.

// Reluctant Steps
// Quantified + Unquantified Pattern Resolution V stringToSearch.
"" + "foo" V "xfooxxxfooa" -> No match
"x" + "foo" V "xfooxxxfooa" -> Match (1) of text "xfoo" starting at index 0 and ending at 4.
"xfoo" from "xfooxxxfooa" consumed
"x" + "foo" V "xxxfooa" -> No match
"xx" + "foo" V "xxxfooa" -> No match
"xxx" + "foo" v "xxxfooa" -> Match (2) of text "xxxfoo" starting at index 4 and ending at
10.
"xxxfoo" from "xxxfooa" consumed leaving "a".
"a" + "foo" V "a" -> No Match
"" + "foo" V "a" -> No Match


// Possessive [+]
regex: .*+foo
stringToSearch: xfooxxxfooa
No match found.

// Possessive steps
// Quantified + Unquantified Pattern Resolution V stringToSearch.
"xfooxxxfooa" + "foo" V "xfooxxxfooa" -> No Match.
```

Showing how eating the string-to-search applies only to that part that hasn't yet been matched by the unquantified pattern.

```
// Greedy [No Symbol]
regex: foo.*
stringToSearch: xfooxxxfooa
Match (1) of text "fooxxxfooa" starting at index 1 and ending at 11.

// Greedy Steps
// Unquantified + Quantified pattern Resolution V stringToSearch.
In "xfooxxxfooa" "foo" matches starting at index 1 and ending at 4.
".*" attempts match against the remainder, "xxxfooa", and succeeds.
Therefore the first match + second match combines to "foo" + "xxxfooa" = "fooxxxfooa"
Processing stops because there are no more characters to consume

// Reluctant [?]
regex: foo.*?
stringToSearch: xfooxxxfooa
Match (1) of text "foo" starting at index 1 and ending at 4.
Match (2) of text "foo" starting at index 7 and ending at 10.

// Reluctant Steps
In "xfooxxxfooa" "foo" matches starting at index 1 and ending at 4.
".*" attempts match against the first position in "xxxfooa", a zero length string. Success.
Therefore the first match + second match combines to "foo" + "" = "foo" (index 1 to 4)

The remaining input string "xxxfooa" is then tested against the whole pattern:
In "xxxfooa" "foo" matches starting at index 7 and ending at 10.
".*" attempts match against the first position in "fooa", a zero length string. Success.
Therefore the first match + second match combines to "foo" + "" = "foo" (index 7 to 10)


// Possessive [+]
regex: foo.*+
stringToSearch: xfooxxxfooa
Match (1) of text "fooxxxfooa" starting at index 1 and ending at 11.
```

```
// Possessive Steps
In "xfooxxxfooa" "foo" matches starting at index 1 and ending at 4.
".*" attempts match against the remainder, "xxxfooa", and succeeds.
Therefore the first match + second match combines to "foo" + "xxxfooa" = "fooxxxfooa"
Processing stops because only one attempt at a match is attempted for a possessive step.
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegeDemo.java

Boundary Matchers

"^" and "$" in regex patterns. If multiline flag not set (default): beginning and end of input. If multiline flag set: beginning and end of line.

```
final static String stringToSearch = "dogcat dog\r\ndog";

// ******* Multiline flag not set (default) *******
// Beginning of input.
regex: ^dog
stringToSearch: dogcat dog
dog
Match (1) of text "dog" starting at index 0 and ending at 3.

// End of input
regex: dog$
stringToSearch: dogcat dog
dog
Match (1) of text "dog" starting at index 12 and ending at 15.


// ******* Multiline flag set *******
// Beginning of the line
regex: ^dog
stringToSearch: dogcat dog
dog
Match (1) of text "dog" starting at index 0 and ending at 3.
Match (2) of text "dog" starting at index 12 and ending at 15.

// End of the line (Mutliline flag set)
regex: dog$
stringToSearch: dogcat dog
dog
Match (1) of text "dog" starting at index 7 and ending at 10.
Match (2) of text "dog" starting at index 12 and ending at 15.
```

Beginning and end of input (the string-to-search), regardless of the multiline flag settings.

```
final static String stringToSearch = "dogcat dog\r\ndog\r\n";


// Beginning of the input
final static String regex = "\\Adog";
regex: \Adog
stringToSearch: dogcat dog
dog
                    // Space from end of line characters.
Match (1) of text "dog" starting at index 0 and ending at 3.


// End if the input but for the final terminator, if any
final static String regex = "dog\\Z";
regex: dog\Z
stringToSearch: dogcat dog
dog
                    // Space from end of line characters.
```

```
Match (1) of text "dog" starting at index 12 and ending at 15.


// End of the input (all input)
final static String regex = "dog\\z";
regex: dog\z
stringToSearch: dogcat dog
dog
                     // Space from end of line characters.
No match found.
```

Word boundary examples.

```
// Word boundary
regex: dog\b
stringToSearch: dogcat dog dog
Match (1) of text "dog" starting at index 7 and ending at 10.
Match (2) of text "dog" starting at index 11 and ending at 14.

// Not a word boundary
regex: dog\B
stringToSearch: dogcat dog dog
Match (1) of text "dog" starting at index 0 and ending at 3.
```

"\G" matches the index of the end of the previous (successful) match. During the first match attempt \G matches the start of the string.

```
// Only the first cat comes after a prior match (before the whole process fails).
final static String regex = "\\Gcat";
final static String stringToSearch = "cat catcat";
regex: \Gcat
stringToSearch: cat catcat
Match (1) of text "cat" starting at index 0 and ending at 3.

// Only the first four letters come after a prior match (before the whole process fails).
final static String regex = "\\G\\w";
final static String stringToSearch = "test me";
regex: \G\w
stringToSearch: test me
Match (1) of text "t" starting at index 0 and ending at 1.
Match (2) of text "e" starting at index 1 and ending at 2.
Match (3) of text "s" starting at index 2 and ending at 3.
Match (4) of text "t" starting at index 3 and ending at 4.
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java based on (Oracle, 2012) http://docs.oracle.com/javase/tutorial/essential/regex/bounds.html and (Goyvaert, 2009) http://www.regular-expressions.info/continue.html

Flags

Flag setting in code.

```
final static String regex = "dog";
final static String stringToSearch = "Dog dog";
final static String replacement = "#";

// Bitwise "or" to set both flags.
final static int flags = Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE;

Pattern pattern = Pattern.compile(BasicRegex.regex, flags);
Matcher matcher = pattern.matcher(stringToSearch);

while (matcher.find()) { ...
```

(Oracle, 2012) http://docs.oracle.com/javase/tutorial/essential/regex/pattern.html

Flag setting in the pattern.

```
final static String regex = "(?iux)dog# comments";
final static String stringToSearch = "Dog dog";
final static String replacement = "@";

// Substitute the matches (java.lang.String)
System.out.printf("Replace all \"%s\" matches of \"%s\" with \"%s\": %s%n", regex,
    stringToSearch, replacement, stringToSearch.replaceAll(regex, replacement));

// Output
Replace all "(?iux)dog# comments" matches of "Dog dog" with "@": @ @
```

(Oracle, 2012) *http://docs.oracle.com/javase/tutorial/essential/regex/pattern.html*

Quotation

Escape metacharacters: with a backslash ("\" or "\\"); enclose between "\Q" and "\E"; or use the Pattern.LITERAL flag (there is no equivalent Embedded Flag).

(Oracle, 2012) *http://docs.oracle.com/javase/tutorial/essential/regex/literals.html*

Quotation example, set in pattern.

```
// With Quoting
final static String regex = "dogpi\\Q[g|h]\\E";
final static String stringToSearch =  "catdogpi[g|h]";

regex: dogpi\Q[g|h]\E
stringToSearch: catdogpi[g|h]
Match (1) of text "dogpi[g|h]" starting at index 3 and ending at 13.
```

Captures

General

Capturing groups are a way to treat one or more characters as a single unit. The capture group can be later referenced: in the current pattern through a backreference ("\"); or in a replacement string with a dollar sign ("$").

Named capture groups, in pattern backreferences or replacement string capture references, is a SE7 Feature.

Capture groups are numbered and identified by counting their opening parentheses from left to right.

```
// Regex Pattern
((A)(B(C))

1.  ((A)(B(C)))
2.  (A)
3.  (B(C))
```

```
4.  (C)

regex: ((\w{3}) (C.{4}(W.*)))
stringToSearch: The CrackWhip
Match (1) of text "The CrackWhip" starting at index 0 and ending at 13.
  Capture groups: 4
  Capture group (0) of text "The CrackWhip" starting at index 0 and ending at 13.
  Capture group (1) of text "The CrackWhip" starting at index 0 and ending at 13.
  Capture group (2) of text "The" starting at index 0 and ending at 3.
  Capture group (3) of text "CrackWhip" starting at index 4 and ending at 13.
  Capture group (4) of text "Whip" starting at index 9 and ending at 13.
replacement: #$4|$3|$2|$1|$0#
replacementResult: #Whip|CrackWhip|The|The CrackWhip|The CrackWhip#
```

(Oracle, 2012) http://docs.oracle.com/javase/tutorial/essential/regex/groups.html

Retrieve the number of captured groups, for a match, with `groupCount()`.

```
final String regex = "(\\w{3})(\\w{3})";
final String stringToSearch = "foobar";

Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(stringToSearch);

// Loop through matches
while (matcher.find()) {
  // Get the number of capture groups for a particular match
  System.out.printf("\tCapture groups: %d%n", matcher.groupCount());
  ...
}
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java

There is also a special group, group 0, which always represents the entire expression. `groupCount()` does not count the special group, group 0.

Non capturing groups do not count towards the `groupCount()` total.

See Java Coding Operations above, page 10, for an example of how capturing groups for each match is coded.

Capture group example

```
regex: (\w{3})(dog)
stringToSearch: catdogpussyratdog
Match (1) of text "catdog" starting at index 0 and ending at 6.
  Capture groups: 2
  Capture group (0) of text "catdog" starting at index 0 and ending at 6.
  Capture group (1) of text "cat" starting at index 0 and ending at 3.
  Capture group (2) of text "dog" starting at index 3 and ending at 6.
Match (2) of text "ratdog" starting at index 11 and ending at 17.
  Capture groups: 2
  Capture group (0) of text "ratdog" starting at index 11 and ending at 17.
  Capture group (1) of text "rat" starting at index 11 and ending at 14.
  Capture group (2) of text "dog" starting at index 14 and ending at 17.
replacement: $2$1
replacementResult: dogcatpussydograt
```

Look ahead and Look behind non-captures

Look ahead and look behind non capture groups effect a prior or subsequent pattern. The look [ahead|behind] non capture group makes the accompanying pattern succeed or fail to match based on the pattern of both groups. The terms "look ahead" and "look behind" refer to not to the non-capture group itself but the accompanying pattern.

```
// Look ahead positive.
regex: (apple|cherry)(?= chocolate)
stringToSearch: Today's specials are apple chocolate pie and cherry banana pie.
Match (1) of text "apple" starting at index 21 and ending at 26.

// Look ahead negative.
regex: (apple|cherry)(?! chocolate)
stringToSearch: Today's specials are apple chocolate pie and cherry banana pie.
Match (1) of text "cherry" starting at index 45 and ending at 51.


// Look behind positive.
regex: (?<=fried )(bananas|clam)
stringToSearch: Tomorrow's special is fried bananas with baked clam.
Match (1) of text "bananas" starting at index 28 and ending at 35.

// Look behind negative
regex: (?<!fried )(bananas|clam)
stringToSearch: Tomorrow's special is fried bananas with baked clam.
Match (1) of text "clam" starting at index 47 and ending at 51.
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java based on (Schreckmann, 2003)

Atomicity (*dodgy section)

Each capture normally consumes characters in the string-to-search ("input string") on each match. However, strings matched by an "atomic" group are not consumed.

The atomic nature of Look ahead and look behinds is exploitable like this.

```
// Matches all words starting with "J" that precede "Schmidt "
// (note the space following the t). The ".+Schmidt " part of the regular expression
// is not consumed because it is in an atomic capture group.
regex: (J\w+)(?=.+Schmidt )
stringToSearch: John Jacob Jingleheimer Schmidt His name is my name, too! Whenever we go
out, The people always shout There goes John Jacob Jingleheimer Schmidt!
Match (1) of text "John" starting at index 0 and ending at 4.
Match (2) of text "Jacob" starting at index 5 and ending at 10.
Match (3) of text "Jingleheimer" starting at index 11 and ending at 23.

// By contrast, observe that a regular (non-atomic) non-capture group consumes
// the charactersin the string-to-search.
regex: (J\w+)(?:.+Schmidt )
stringToSearch: John Jacob Jingleheimer Schmidt His name is my name, too! Whenever we go
out, The people always shout There goes John Jacob Jingleheimer Schmidt!
Match (1) of text "John Jacob Jingleheimer Schmidt " starting at index 0 and ending at 32.
  Capture groups: 1
  Capture group (0) of text "John Jacob Jingleheimer Schmidt " starting at index 0
    and ending at 32.
  Capture group (1) of text "John" starting at index 0 and ending at 4.
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java based on (Schreckmann, 2003), Look Ahead and Look Behind Constructs, Search for "Jim Yingst pointed out an important issue ..."

An independent non-capture group is a non-capture group that is atomic.

```
// Normal matching
regex: a(bc|b)c
stringToSearch: abccabc
Match (1) of text "abcc" starting at index 0 and ending at 4.
  Capture groups: 1
  Capture group (0) of text "abcc" starting at index 0 and ending at 4.
  Capture group (1) of text "bc" starting at index 1 and ending at 3.
Match (2) of text "abc" starting at index 4 and ending at 7.
  Capture groups: 1
  Capture group (0) of text "abc" starting at index 4 and ending at 7.
  Capture group (1) of text "b" starting at index 5 and ending at 6.

// Normal non-capturing group
regex: a(?:bc|b)c
stringToSearch: abccabc
Match (1) of text "abcc" starting at index 0 and ending at 4.
  Capture groups: 0
  Capture group (0) of text "abcc" starting at index 0 and ending at 4.
Match (2) of text "abc" starting at index 4 and ending at 7.
  Capture groups: 0
  Capture group (0) of text "abc" starting at index 4 and ending at 7.

// Independent (atomic) non-capturing group
regex: a(?>bc|b)c
stringToSearch: abccabc
Match (1) of text "abcc" starting at index 0 and ending at 4.
  Capture groups: 0
  Capture group (0) of text "abcc" starting at index 0 and ending at 4.
```

Backreferences

A backreference **occurs in a regex pattern** and refers to a sub-sequence in the string-to-Search matched by a previous capture group. In effect they identify repeating groups of characters.

The syntax is backslash followed by a number from 1 to 9 (e.g. "\1") but in Java Strings these are escaped (e.g. "\\1").

A zero ("\0") backreference throws an exception (you can't remember the whole match before you've processed it).

Backreference example.

```
final static String regex = "(\\w{3})(\\1)";

regex: (\w{3})(\1)
stringToSearch: catdogcatcat
Match (1) of text "catcat" starting at index 6 and ending at 12.
  Capture groups: 2
  Capture group (0) of text "catcat" starting at index 6 and ending at 12.
  Capture group (1) of text "cat" starting at index 6 and ending at 9.
  Capture group (2) of text "cat" starting at index 9 and ending at 12.
```

"catdog" not matched: the backreference is not simply referring to the kind of match. That is, in this example when the backtrack "\1" references "(\w{3})" the backtrack is **not** asking "Match any three word characters again."
"catcat" is matched: the backreference references the actual match ("cat") of the capture group "(\w{3})".

(Oracle, 2012) *http://docs.oracle.com/javase/tutorial/essential/regex/groups.html*

Backreferences reference a capture group, not a mere prior match.

```
// Works
final static String regex = "(\\d{2})\\1";

regex: (\d{2})\1
stringToSearch: AAAA 9999
Match (1) of text "9999" starting at index 5 and ending at 9.
  Capture groups: 1
  Capture group (0) of text "9999" starting at index 5 and ending at 9.
  Capture group (1) of text "99" starting at index 5 and ending at 7.
replacement: @
replacementResult: AAAA @

// Doesn't work
final static String regex = "\\d{2}\\1";

regex: \d{2}\1
stringToSearch: AAAA 9999
No match found.
```

Named backreferences are a SE7 feature.

Replacement

Replacement strings can contain references to capture groups in the form of: `$g`, where `g = 0 to 9`. Refer to the special capture group, which might equal a match, with `$0`.

```
regex: ([cr]at)(.at)
stringToSearch: catdogratbat
Match (1) of text "ratbat" starting at index 6 and ending at 12.
  Capture groups: 2
  Capture group (0) of text "ratbat" starting at index 6 and ending at 12.
  Capture group (1) of text "rat" starting at index 6 and ending at 9.
  Capture group (2) of text "bat" starting at index 9 and ending at 12.
replacement: #$2|$1|$0#
replacementResult: catdog#bat|rat|ratbat#
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java

Note that by default the replacement string replaces the special capture group ($0) which, in effect replaces a match.

```
// Replace the special capture group by default
regex: cat
stringToSearch: pigcatdog
Match (1) of text "cat" starting at index 3 and ending at 6.
  Capture groups: 0
  Capture group (0) of text "cat" starting at index 3 and ending at 6.
replacement: @
replacementResult: pig@dog

// Replace the special capture group with itself
regex: cat
stringToSearch: pigcatdog
Match (1) of text "cat" starting at index 3 and ending at 6.
  Capture groups: 0
  Capture group (0) of text "cat" starting at index 3 and ending at 6.
replacement: $0
replacementResult: pigcatdog
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java

Named capture groups, `${name}`, is a SE7 feature.

Java Single line `ReplaceAll()`.

```
// ... from previous example.

// Substitute the matches (java.lang.String)
System.out.printf("Replace all \"%s\" matches of \"%s\" with \"%s\": %s%n", regex,
    stringToSearch, replacement, stringToSearch.replaceAll(regex, replacement));

// Substitute the matches (java.util.regex)
System.out.printf("Replace all \"%s\" matches of \"%s\" with \"%s\": %s%n", regex,
    stringToSearch, replacement, Pattern.compile(regex).matcher(stringToSearch)
        .replaceAll(replacement));

// Output for either
Replace all "([cr]at)(.at)" matches of "catdogratbat" with "#$2|$1|$0#":
catdog#bat|rat|ratbat#
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java

Unicode Support

For Unicode Support in the Java Language in general:

See (Bentley, Java Reference - Language.dox, 2013), Strings > Unicode Representations

Unicode Support in Java Regex's occurs in several ways: native support in strings; reference a character by their Unicode Code Point in hexadecimal; or reference a Unicode character set through scripts, blocks, categories and binary properties.

```
// Code ...

// \u03B2 is the Unicode Code Point for the greek letter beta "β".
// \u5229 is the Unicode Code Point for the Hiragana letter "利".
final static String regex = "(利)|(\u03B2)|\\p{InHiragana}";
final static String stringToSearch = "Hello 利\u5229 Greek \u03B2";
final static String replacement = "@";


// Output ...

// How the regex pattern resolves when output.
regex: (利)|(β)|\p{InHiragana}

// How the stringToSearch resolves when output.
stringToSearch: Hello 利利 Greek β

Match (1) of text "利" starting at index 6 and ending at 7.
Match (2) of text "利" starting at index 7 and ending at 8.
Match (3) of text "β" starting at index 15 and ending at 16.

replacement: @
replacementResult: Hello @@ Greek @
```

(Bentley, TutorialAtOracle Code Examples - Regex, 2013) RegexDemo.java
See also above, Pattern ("regex") Syntax

For the set of legal Unicode Character Sets defined through scripts, blocks, categories and binary properties follow the links ...

... above, Character sets, "Character sets for Unicode scripts, blocks, categories and binary properties"

References, Word

Bentley, J. (2013, Jul 13). Java Reference - Language.dox.

Bentley, J. (2013, Jul). TutorialAtOracle Code Examples - Regex. Retrieved from C:\Users\John\Documents\Sda\Code\Java\Examples\TutorialAtOracle\HelloWorld

Goyvaert, J. (2009, Jun 17). Retrieved Jul 10, 2013, from regular-expressions.info: http://www.regular-expressions.info/

Oracle. (2011). *Java Platform, Standard Edition 6, API Specification*. Retrieved Jul 09, 2013, from http://docs.oracle.com: http://docs.oracle.com/javase/6/docs/api/overview-summary.html

Oracle. (2012). *The Java Tutorials.* Retrieved Jun 14, 2012, from http://docs.oracle.com/: http://docs.oracle.com/javase/tutorial/

Schreckmann, D. (2003, Mar 31). *An Introduction to java.util.regex, Part 2: More Pattern Elements*. Retrieved Jul 10, 2013, from Java Ranch: http://www.javaranch.com/journal/2003/04/RegexTutorial.htm

Document Licence