Java - Regular Expressions

Java provides the java.util.regex package for pattern matching with regular expressions. Java regular expressions are very similar to the Perl programming language and very easy to learn.

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. They can be used to search, edit, or manipulate text and data.

The java.util.regex package primarily consists of the following three classes −

- **Pattern Class** − A Pattern object is a compiled representation of a regular expression. The Pattern class provides no public constructors. To create a pattern, you must first invoke one of its public static **compile()** methods, which will then return a Pattern object. These methods accept a regular expression as the first argument.
- **Matcher Class** − A Matcher object is the engine that interprets the pattern and performs match operations against an input string. Like the Pattern class, Matcher defines no public constructors. You obtain a Matcher object by invoking the **matcher()** method on a Pattern object.
- **PatternSyntaxException** − A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

Capturing Groups

Capturing groups are a way to treat multiple characters as a single unit. They are created by placing the characters to be grouped inside a set of parentheses. For example, the regular expression (dog) creates a single group containing the letters "d", "o", and "g".

Capturing groups are numbered by counting their opening parentheses from the left to the right. In the expression ((A)(B(C))), for example, there are four such groups −

- ((A)(B(C)))
- (A)
- (B(C))
- (C)

To find out how many groups are present in the expression, call the groupCount method on a matcher object. The groupCount method returns an **int** showing the number of capturing groups present in the matcher's pattern.

There is also a special group, group 0, which always represents the entire expression. This group is not included in the total reported by groupCount.

Example

Following example illustrates how to find a digit string from the given alphanumeric string −

Live Demo

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

public class RegexMatches {

   public static void main( String args[] ) {
      // String to be scanned to find the pattern.
      String line = "This order was placed for QT3000! OK?";
      String pattern = "(.*)(\\d+)(.*)";

      // Create a Pattern object
      Pattern r = Pattern.compile(pattern);

      // Now create matcher object.
      Matcher m = r.matcher(line);
      if (m.find( )) {
         System.out.println("Found value: " + m.group(0) );
         System.out.println("Found value: " + m.group(1) );
         System.out.println("Found value: " + m.group(2) );
      }else {
         System.out.println("NO MATCH");
      }
   }
}
```

This will produce the following result −

Output

```
Found value: This order was placed for QT3000! OK?
Found value: This order was placed for QT300
Found value: 0
```

Regular Expression Syntax

Here is the table listing down all the regular expression metacharacter syntax available in Java −

Subexpression	Matches
^	Matches the beginning of the line.
$	Matches the end of the line.
.	Matches any single character except newline. Using **m** option allows it to match the newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets.
\A	Beginning of the entire string.
\z	End of the entire string.
\Z	End of the entire string except allowable final line terminator.
re*	Matches 0 or more occurrences of the preceding expression.
re+	Matches 1 or more of the previous thing.
re?	Matches 0 or 1 occurrence of the preceding expression.
re{ n}	Matches exactly n number of occurrences of the preceding expression.
re{ n,}	Matches n or more occurrences of the preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of the preceding expression.
a| b	Matches either a or b.
(re)	Groups regular expressions and remembers the matched text.
(?: re)	Groups regular expressions without remembering the matched text.
(?> re)	Matches the independent pattern without backtracking.
\w	Matches the word characters.
\W	Matches the nonword characters.
\s	Matches the whitespace. Equivalent to [\t\n\r\f].
\S	Matches the nonwhitespace.
\d	Matches the digits. Equivalent to [0-9].
\D	Matches the nondigits.
\A	Matches the beginning of the string.

\Z	Matches the end of the string. If a newline exists, it matches just before newline.
\z	Matches the end of the string.
\G	Matches the point where the last match finished.
\n	Back-reference to capture group number "n".
\b	Matches the word boundaries when outside the brackets. Matches the backspace (0x08) when inside the brackets.
\B	Matches the nonword boundaries.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\Q	Escape (quote) all characters up to \E.
\E	Ends quoting begun with \Q.

Methods of the Matcher Class

Here is a list of useful instance methods −

Index Methods

Index methods provide useful index values that show precisely where the match was found in the input string −

Sr.No.	Method & Description
1	**public int start()** Returns the start index of the previous match.
2	**public int start(int group)** Returns the start index of the subsequence captured by the given group during the previous match operation.
3	**public int end()** Returns the offset after the last character matched.
4	**public int end(int group)** Returns the offset after the last character of the subsequence captured by the given group during the previous match operation.

Study Methods

Study methods review the input string and return a Boolean indicating whether or not the pattern is found −

Sr.No.	Method & Description
1	**public boolean lookingAt()** Attempts to match the input sequence, starting at the beginning of the region, against the pattern.
2	**public boolean find()** Attempts to find the next subsequence of the input sequence that matches the pattern.
3	**public boolean find(int start)** Resets this matcher and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index.
4	**public boolean matches()** Attempts to match the entire region against the pattern.

Replacement Methods

Replacement methods are useful methods for replacing text in an input string −

Sr.No.	Method & Description
1	**public Matcher appendReplacement(StringBuffer sb, String replacement)** Implements a non-terminal append-and-replace step.
2	**public StringBuffer appendTail(StringBuffer sb)** Implements a terminal append-and-replace step.
3	**public String replaceAll(String replacement)** Replaces every subsequence of the input sequence that matches the pattern with the given replacement string.
4	**public String replaceFirst(String replacement)** Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.
5	**public static String quoteReplacement(String s)** Returns a literal replacement String for the specified String. This method produces a String that will work as a literal replacement **s** in the appendReplacement method of the Matcher class.

The start and end Methods

Following is the example that counts the number of times the word "cat" appears in the input string −

Example

Live Demo

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

public class RegexMatches {

   private static final String REGEX = "\\bcat\\b";
   private static final String INPUT = "cat cat cat cattie cat";

   public static void main( String args[] ) {
      Pattern p = Pattern.compile(REGEX);
      Matcher m = p.matcher(INPUT);   // get a matcher object
      int count = 0;

      while(m.find()) {
         count++;
         System.out.println("Match number "+count);
         System.out.println("start(): "+m.start());
         System.out.println("end(): "+m.end());
      }
   }
}
```

This will produce the following result −

Output

```
Match number 1
start(): 0
end(): 3
Match number 2
start(): 4
end(): 7
Match number 3
start(): 8
end(): 11
Match number 4
start(): 19
end(): 22
```

You can see that this example uses word boundaries to ensure that the letters "c" "a" "t" are not merely a substring in a longer word. It also gives some useful information about where in the input string the match has occurred.

The start method returns the start index of the subsequence captured by the given group during the previous match operation, and the end returns the index of the last character matched, plus one.

The matches and lookingAt Methods

The matches and lookingAt methods both attempt to match an input sequence against a pattern. The difference, however, is that matches requires the entire input sequence to be matched, while lookingAt does not.

Both methods always start at the beginning of the input string. Here is the example explaining the functionality −

Example

Live Demo

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

public class RegexMatches {

   private static final String REGEX = "foo";
   private static final String INPUT = "fooooooooooooooooo";
   private static Pattern pattern;
   private static Matcher matcher;

   public static void main( String args[] ) {
      pattern = Pattern.compile(REGEX);
      matcher = pattern.matcher(INPUT);

      System.out.println("Current REGEX is: "+REGEX);
      System.out.println("Current INPUT is: "+INPUT);

      System.out.println("lookingAt(): "+matcher.lookingAt());
      System.out.println("matches(): "+matcher.matches());
   }
}
```

This will produce the following result −

Output

```
Current REGEX is: foo
Current INPUT is: fooooooooooooooooo
lookingAt(): true
matches(): false
```

The replaceFirst and replaceAll Methods

The replaceFirst and replaceAll methods replace the text that matches a given regular expression. As their names indicate, replaceFirst replaces the first occurrence, and replaceAll replaces all occurrences.

Here is the example explaining the functionality −

Example

Live Demo

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

public class RegexMatches {

   private static String REGEX = "dog";
   private static String INPUT = "The dog says meow. " + "All dogs say meow.";
   private static String REPLACE = "cat";

   public static void main(String[] args) {
      Pattern p = Pattern.compile(REGEX);

      // get a matcher object
      Matcher m = p.matcher(INPUT);
      INPUT = m.replaceAll(REPLACE);
      System.out.println(INPUT);
   }
}
```

This will produce the following result −

Output

```
The cat says meow. All cats say meow.
```

The appendReplacement and appendTail Methods

The Matcher class also provides appendReplacement and appendTail methods for text replacement.

Here is the example explaining the functionality −

Example

Live Demo

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

public class RegexMatches {

   private static String REGEX = "a*b";
   private static String INPUT = "aabfooaabfooabfoob";
   private static String REPLACE = "-";
   public static void main(String[] args) {

      Pattern p = Pattern.compile(REGEX);

      // get a matcher object
      Matcher m = p.matcher(INPUT);
      StringBuffer sb = new StringBuffer();
      while(m.find()) {
         m.appendReplacement(sb, REPLACE);
      }
      m.appendTail(sb);
      System.out.println(sb.toString());
   }
}
```

This will produce the following result −

Output

```
-foo-foo-foo-
```

PatternSyntaxException Class Methods

A PatternSyntaxException is an unchecked exception that indicates a syntax error in a regular expression pattern. The PatternSyntaxException class provides the following methods to help you determine what went wrong −

Sr.No.	Method & Description
1	**public String getDescription()** Retrieves the description of the error.
2	**public int getIndex()** Retrieves the error index.
3	**public String getPattern()** Retrieves the erroneous regular expression pattern.
4	**public String getMessage()** Returns a multi-line string containing the description of the syntax error and its index, the erroneous regular expression pattern, and a visual indication of the error index within the pattern.