Java Regex - Java Regular Expressions

Java regex is the official Java regular expression API. The term *Java regex* is an abbreviation of *Java regular expression*. The Java regex API is located in the `java.util.regex` package which has been part of standard Java (JSE) since Java 1.4. This Java regex tutorial will explain how to use this API to match regular expressions against text.

Although Java regex has been part of standard Java since Java 1.4, this Java regex tutorial covers the Java regex API released with Java 8.

Regular Expressions

A regular expression is a textual pattern used to search in text. You do so by "matching" the regular expression against the text. The result of matching a regular expression against a text is either:

- A `true` / `false` specifying if the regular expression matched the text.
- A set of matches - one match for every occurrence of the regular expression found in the text.

For instance, you could use a regular expression to search an **Java String** for email addresses, URLs, telephone numbers, dates etc. This would be done by matching different regular expressions against the String. The result of matching each regular expression against the String would be a set of matches - one set of matches for each regular expression (each regular expression may match more than one time).

I will show you some examples of how to match regular expressions against text with the Java regex API further down this page. But first I will introduce the core classes of the Java regex API in the following section.

Java Regex Core Classes

The Java regex API consists of two core classes. These are:

- `Pattern (java.util.regex.Pattern)`
- `Matcher (java.util.regex.Matcher)`

The `Pattern` class is used to create patterns (regular expressions). A pattern is precompiled regular expression in object form (as a `Pattern` instance), capable of matching itself against a text.

The `Matcher` class is used to match a given regular expression (`Pattern` instance) against a text multiple times. In other words, to look for multiple occurrences of the regular expression in the text. The `Matcher` will tell you where in the text (character index) it found the occurrences. You can obtain a `Matcher` instance from a `Pattern` instance.

Both the `Pattern` and `Matcher` classes are covered in detail in their own texts. See links above, or in the top left of every page in this Java regex tutorial trail.

Java Regular Expression Example

As mentioned above the Java regex API can either tell you if a regular expression matches a certain String, or return all the matches of that regular expression in the String. The following sections will show you examples of both of these ways to use the Java regex API.

Pattern Example

Here is a simple java regex example that uses a regular expression to check if a text contains the substring `http://` :

```
String text    =
        "This is the text to be searched " +
        "for occurrences of the http:// pattern.";

String regex = ".*http://.*";

boolean matches = Pattern.matches(regex, text);

System.out.println("matches = " + matches);
```

The `text` variable contains the text to be checked with the regular expression.

The `pattern` variable contains the regular expression as a `String`. The regular expression matches all texts which contains one or more characters (`.*`) followed by the text `http://`followed by one or more characters (`.*`).

The third line uses the `Pattern.matches()` static method to check if the regular expression (pattern) matches the text. If the regular expression matches the text, then `Pattern.matches()` returns true. If the regular expression does not match the text `Pattern.matches()` returns false.

The example does not actually check if the found `http://` string is part of a valid URL, with domain name and suffix (.com, .net etc.). The regular expression just checks for an occurrence of the string `http://`.

Matcher Example

Here is another Java regex example which uses the `Matcher` class to locate multiple occurrences of the substring "is" inside a text:

```
String text    =
        "This is the text which is to be searched " +
        "for occurrences of the word 'is'.";

String regex = "is";

Pattern pattern = Pattern.compile(regex);
```

```
Matcher matcher = pattern.matcher(text);

int count = 0;
while(matcher.find()) {
    count++;
    System.out.println("found: " + count + " : "
            + matcher.start() + " - " + matcher.end());
}
```

From the `Pattern` instance a `Matcher` instance is obtained. Via this `Matcher` instance the example finds all occurrences of the regular expression in the text.

Java Regular Expression Syntax

A key aspect of regular expressions is the regular expression syntax. Java is not the only programming language that has support for regular expressions. Most modern programming languages supports regular expressions. The syntax used in each language define regular expressions is not exactly the same, though. Therefore you will need to learn the syntax used by your programming language.

In the following sections of this Java regex tutorial I will give you examples of the Java regular expression syntax, to get you started with the Java regex API and regular expressions in general. The regular expression syntax used by the Java regex API is covered in detail in the text about the **Java regular expression syntax**

Matching Characters

The first thing to look at is how to write a regular expression that matches characters against a given text. For instance, the regular expression defined here:

```
String regex = "http://";
```

will match all strings that are exactly the same as the regular expression. There can be no characters before or after the `http://` - or the regular expression will not match the text. For instance, the above regex will match this text:

```
String text1 = "http://";
```

But not this text:

```
String text2 = "The URL is: http://mydomain.com";
```

The second string contains characters both before and after the `http://` that is matched against.

Metacharacters

Metacharacters are characters in a regular expression that are interpreted to have special meanings. These metacharacters are:

Character	**Description**
<	
>	
(	
)	
[	
]	
{	
}	
\	
^	
-	
=	
$	
!	
|	
?	
*	
+	
.	

What exactly these metacharacters mean will be explained further down this Java Regex tutorial. Just keep in mind that if you include e.g. a "." (fullstop) in a regular expression it will not match a fullstop character, but match something else which is defined by that metacharacter (also explained later).

Escaping Characters

As mentioned above, metacharacters in Java regular expressions have a special meaning. If you really want to match these characters in their literal form, and not their metacharacter meaning, you must "escape" the metacharacer you want to match. To escape a metacharacter you use the Java regular expression escape character - the backslash character. Escaping a character means preceding it with the backslash character. For instance, like this:

```
\.
```

In this example the `.` character is preceded (escaped) by the `\` character. When escaped the fullstop character will actually match a fullstop character in the input text. The special metacharacter meaning of an escaped metacharacter is ignored - only its actual literal value (e.g. a fullstop) is used.

Java regular expression syntax uses the backslash character as escape character, just like Java Strings do. This gives a little challenge when writing a regular expression in a Java string. Look at this regular expression example:

```
String regex = "\\.";
```

Notice that the regular expression String contains two backslashes after each other, and then a `.` . The reason is, that first the Java compiler interprets the two `\\` characters as an escaped Java String character. After the Java compiler is done, only one `\` is left, as `\\` means the character `\`. The string thus looks like this:

```
\.
```

Now the Java regular expression interpreter kicks in, and interprets the remaining backslash as an escape character. The following character `.` is now interpreted to mean an actual full stop, not to have the special regular expression meaning it otherwise has. The remaining regular expression thus matches for the full stop character and nothing more.

Several characters have a special meaning in the Java regular expression syntax. If you want to match for that explicit character and not use it with its special meaning, you need to escape it with the backslash character first. For instance, to match for the full stop character, you need to write:

```
String regex = "\\.";
```

To match for the backslash character itself, you need to write:

```
String regex = "\\\\";
```

Getting the escaping of characters right in regular expressions can be tricky. For advanced regular expressions you might have to play around with it a while before you get it right.

Matching Any Character

So far we have only seen how to match specific characters like "h", "t", "p" etc. However, you can also just match any character without regard to what character it is. The Java regular expression syntax lets you do that using the `.` character (period / full stop). Here is an example regular expression that matches any character:

```
String regex = ".";
```

This regular expression matches a single character, no matter what character it is.

The `.` character can be combined with other characters to create more advanced regular expressions. Here is an example:

```
String regex = "H.llo";
```

This regular expression will match any Java string that contains the characters "H" followed by any character, followed by the characters "llo". Thus, this regular expression will match all of the strings "Hello", "Hallo", "Hullo", "Hxllo" etc.

Matching Any of a Set of Characters

Java regular expressions support matching any of a specified set of characters using what is referred to as character classes. Here is a character class example:

```
String regex = "H[ae]llo";
```

The character class (set of characters to match) is enclosed in the square brackets - the `[ae]` part of the regular expression, in other words. The square brackets are not matched - only the characters inside them.

The character class will match one of the enclosed characters regardless of which, but no mor than one. Thus, the regular expression above will match any of the two strings "Hallo" or "Hello", but no other strings. Only an "a" or an "e" is allowed between the "H" and the "llo".

You can match a range of characters by specifying the first and the last character in the range with a dash in between. For instance, the character class `[a-z]` will match all characters between a lowercase `a` and a lowercase `z`, both `a` and `z` included.

You can have more than one character range within a character class. For instance, the character class `[a-zA-Z]` will match all letters between `a` and `z` or between `A` and `Z` .

You can also use ranges for digits. For instance, the character class `[0-9]` will match the characters between 0 and 9, both included.

If you want to actually match one of the square brackets in a text, you will need to escape them. Here is how escaping the square brackets look:

```
String regex = "H\\[llo";
```

The `\\[` is the escaped square left bracket. This regular expression will match the string "H[llo".

If you want to match the square brackets inside a character class, here is how that looks:

```
String regex = "H[\\[\\]]llo";
```

The character class is this part: `[\\[\\]]`. The character class contains the two square brackets escaped (`\\[` and `\\]`).

This regular expression will match the strings "H[llo" and "H]llo".

Matching a Range of Characters

The Java regex API allows you to specify a range of characters to match. Specifying a range of characters is easier than explicitly specifying each character to match. For instance, you can match the characters a to z like this:

```
String regex = "[a-z]";
```

This regular expression will match any single character from a to z in the alphabet.

The character classes are case sensitive. To match all characters from a to z regardless of case, you must include both uppercase and lowercase character ranges. Here is how that looks:

```
String regex = "[a-zA-Z]";
```

Matching Digits

You can match digits of a number with the predefined character class with the code `\d`. The digit character class corresponds to the character class `[0-9]`.

Since the `\` character is also an escape character in Java, you need two backslashes in the Java string to get a `\d` in the regular expression. Here is how such a regular expression string looks:

```
String regex = "Hi\\d";
```

This regular expression will match strings starting with "Hi" followed by a digit (`0` to `9`). Thus, it will match the string "Hi5" but not the string "Hip".

Matching Non-digits

Matching non-digits can be done with the predefined character class `[\D]` (uppercase D). Here is an regular expression containing the non-digit character class:

```
String regex = "Hi\\D";
```

This regular expression will match any string which starts with "Hi" followed by one character which is not a digit.

Matching Word Characters

You can match word characters with the predefined character class with the code `\w` . The word character class corresponds to the character class `[a-zA-Z_0-9]`.

```
String regex = "Hi\\w";
```

This regular expression will match any string that starts with "Hi" followed by a single word character.

Matching Non-word Characters

You can match non-word characters with the predefined character class `[\W]` (uppercase W). Since the `\` character is also an escape character in Java, you need two backslashes in the Java string to get a `\w` in the regular expression. Here is how such a regular expression string looks:

Here is a regular expression example using the non-word character class:

```
String regex = "Hi\\w";
```

Boundaries

The Java Regex API can also match *boundaries* in a string. A boundary could be the beginning of a string, the end of a string, the beginning of a word etc. The Java Regex API supports the following boundaries:

The end of the input

Symbol	Description
^	The beginning of a line.
$	The end of a line.
\b	A word boundary (where a word starts or ends, e.g. space, tab etc.).
\B	A non-word boundary.
\A	The beginning of the input.
\G	The end of the previous match.
\Z	The end of the input but for the final terminator (if any).
\z	

Some of these boundary matchers are explained below.

Beginning of Line (or String)

The `^` boundary matcher matches the beginning of a line according to the Java API specification. However, in practice it seems to only be matching the beginning of a String. For instance, the following example only gets a single match at index 0:

```
String text = "Line 1\nLine2\nLine3";

Pattern pattern = Pattern.compile("^");
Matcher matcher = pattern.matcher(text);

while(matcher.find()){
    System.out.println("Found match at: "  + matcher.start() + " to " + matcher.end());
}
```

Even if the input string contains several line breaks, the `^` character only matches the beginning of the input string, not the beginning of each line (after each line break).

The beginning of line / string matcher is often used in combination with other characters, to check if a string begins with a certain substring. For instance, this example checks if the input string starts with the substring `http://` :

```
String text = "http://jenkov.com";

Pattern pattern = Pattern.compile("^http://");
Matcher matcher = pattern.matcher(text);

while(matcher.find()){
    System.out.println("Found match at: "  + matcher.start() + " to " + matcher.end());
}
```

This example finds a single match of the substring `http://` from index 0 to index 7 in the input stream. Even if the input string had contained more instances of the substring`http://` they would not have been matched by this regular expression, since the regular expression started with the `^` character.

End of Line (or String)

The `$` boundary matcher matches the end of the line according to the Java specification. In practice, however, it looks like it only matches the end of the input string.

The beginning of line (or string) matcher is often used in combination with other characters, most commonly to check if a string ends with a certain substring. Here is an example of the end of line / string matcher:

```
String text = "http://jenkov.com";

Pattern pattern = Pattern.compile(".com$");
Matcher matcher = pattern.matcher(text);

while(matcher.find()){
    System.out.println("Found match at: "  + matcher.start() + " to " + matcher.end());
}
```

This example will find a single match at the end of the input string.

Word Boundaries

The `\b` boundary matcher matches a word boundary, meaning a location in an input string where a word either starts or ends.

Here is a Java regex word boundary example:

```
String text = "Mary had a little lamb";

Pattern pattern = Pattern.compile("\\b");
Matcher matcher = pattern.matcher(text);

while(matcher.find()){
    System.out.println("Found match at: "  + matcher.start() + " to " + matcher.end());
}
```

This example matches all word boundaries found in the input string. Notice how the word boundary matcher is written as `\\b` - with two `\\` (backslash) characters. The reason for this is explained in the section about **escaping characters**. The Java compiler uses `\` as an escape character, and thus requires two backslash characters after each other in order to insert a single backslash character into the string.

The output of running this example would be:

```
Found match at: 0 to 0
Found match at: 4 to 4
Found match at: 5 to 5
Found match at: 8 to 8
Found match at: 9 to 9
Found match at: 10 to 10
Found match at: 11 to 11
Found match at: 17 to 17
Found match at: 18 to 18
Found match at: 22 to 22
```

The output lists all the locations where a word either starts or ends in the input string. As you can see, the indices of word beginnings point to the first character of the word, whereas endings of a word points to the first character after the word.

You can combine the word boundary matcher with other characters to search for words beginning with specific characters. Here is an example:

```
String text = "Mary had a little lamb";

Pattern pattern = Pattern.compile("\\bl");
Matcher matcher = pattern.matcher(text);

while(matcher.find()){
    System.out.println("Found match at: "  + matcher.start() + " to " + matcher.end());
}
```

This example will find all the locations where a word starts with the letter `l` (lowercase). In fact it will also find the ends of these matches, meaning the last character of the pattern, which is the lowercase `l` letter.

Non-word Boundaries

The `\B` boundary matcher matches non-word boundaries. A non-word boundary is a boundary between two characters which are both part of the same word. In other words, the character combination is not word-to-non-word character sequence (which is a word boundary). Here is a simple Java regex non-word boundary matcher example:

```
String text = "Mary had a little lamb";

Pattern pattern = Pattern.compile("\\B");
Matcher matcher = pattern.matcher(text);

while(matcher.find()){
    System.out.println("Found match at: "  + matcher.start() + " to " + matcher.end());
}
```

This example will give the following output:

```
Found match at: 1 to 1
Found match at: 2 to 2
Found match at: 3 to 3
Found match at: 6 to 6
Found match at: 7 to 7
Found match at: 12 to 12
Found match at: 13 to 13
Found match at: 14 to 14
Found match at: 15 to 15
Found match at: 16 to 16
Found match at: 19 to 19
Found match at: 20 to 20
Found match at: 21 to 21
```

Notice how these match indexes corresponds to boundaries between characters within the same word.

Quantifiers

Quantifiers can be used to match characters more than once. There are several types of quantifiers which are listed in the **Java Regex Syntax**. I will introduce some of the most commonly used quantifiers here.

The first two quantifiers are the `*` and `+` characters. You put one of these characters after the character you want to match multiple times. Here is a regular expression with a quantifier:

```
String regex = "Hello*";
```

This regular expression matches strings with the text "Hell" followed by zero or more `o`characters. Thus, the regular expression will match "Hell", "Hello", "Helloo" etc.

If the quantifier had been the `+` character instead of the `*` character, the string would have had to end with 1 or more `o` characters.

If you want to match any of the two quantifier characters you will need to escape them. Here is an example of escaping the `+` quantifier:

```
String regex = "Hell\\+";
```

This regular expression will match the string "Hell+";

You can also match an exact number of a specific character using the `{n}` quantifier, where `n` is the number of characters you want to match. Here is an example:

```
String regex = "Hello{2}";
```

This regular expression will match the string "Helloo" (with two `o` characters in the end).

You can set an upper and a lower bound on the number of characters you want to match, like this:

```
String regex = "Hello{2,4}";
```

This regular expression will match the strings "Helloo", "Hellooo" and "Helloooo". In other words, the string "Hell" with 2, 3 or 4 `o` characters in the end.

Java String Regex Methods

The Java String class has a few regular expression methods too. I will cover some of those here:

matches()

The Java String `matches()` method takes a regular expression as parameter, and returns `true` if the regular expression matches the string, and `false` if not.

Here is a `matches()` example:

```
String text = "one two three two one";

boolean matches = text.matches(".*two.*");
```

split()

The Java String `split()` method splits the string into N substrings and returns a String array with these substrings. The `split()` method takes a regular expression as parameter and splits the string at all positions in the string where the regular expression matches a part of the string. The regular expression is not returned as part of the returned substrings.

Here is a `split()` example:

```
String text = "one two three two one";

String[] twos = text.split("two");
```

This example will return the three strings "one", " three" and " one".

replaceFirst()

The Java String `replaceFirst()` method returns a new String with the first match of the regular expression passed as first parameter with the string value of the second parameter.

Here is a `replaceFirst()` example:

```
String text = "one two three two one";

String s = text.replaceFirst("two", "five");
```

This example will return the string "one five three two one".

replaceAll()

The Java String `replaceAll()` method returns a new String with all matches of the regular expression passed as first parameter with the string value of the second parameter.

Here is a `replaceAll()` example:

```
String text = "one two three two one";

String t = text.replaceAll("two", "five");
```

This example will return the string "one five three five one".

Java Regex - Pattern

The Java `Pattern` class (`java.util.regex.Pattern`), is the main access point of the Java regular expression API. Whenever you need to work with regular expressions in Java, you start with Java's `Pattern` class.

Working with regular expressions in Java is also sometimes referred to as *pattern matching in Java*. A regular expression is also sometimes referred to as a *pattern*(hence the name of the Java `Pattern` class). Thus, the term pattern matching in Java means matching a regular expression (pattern) against a text using Java.

The Java `Pattern` class can be used in two ways. You can use the `Pattern.matches()`method to quickly check if a text (String) matches a given regular expression. Or you can compile a `Pattern` instance using `Pattern.compile()` which can be used multiple times to match the regular expression against multiple texts. Both the `Pattern.matches()` and `Pattern.compile()` methods are covered below.

Pattern.matches()

The easiest way to check if a regular expression pattern matches a text is to use the static `Pattern.matches()` method. Here is a `Pattern.matches()` example in Java code:

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

public class PatternMatchesExample {

    public static void main(String[] args) {

        String text    =
            "This is the text to be searched " +
            "for occurrences of the pattern.";

        String pattern = ".*is.*";

        boolean matches = Pattern.matches(pattern, text);

        System.out.println("matches = " + matches);
    }
}
```

This `Pattern.matches()` example searches the string referenced by the `text` variable for an occurrence of the word "is", allowing zero or more characters to be present before and after the word (the two `.*` parts of the pattern).

The `Pattern.matches()` method is fine if you just need to check a pattern against a text a single time, and the default settings of the `Pattern` class are appropriate.

If you need to match for multiple occurrences, and even access the various matches, or just need non-default settings, you need to compile a `Pattern` instance using the `Pattern.compile()` method.

Pattern.compile()

If you need to match a text against a regular expression pattern more than one time, you need to create a `Pattern` instance using the `Pattern.compile()` method. Here is a Java `Pattern.compile()` example:

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

public class PatternCompileExample {

    public static void main(String[] args) {

        String text    =
                "This is the text to be searched " +
                "for occurrences of the http:// pattern.";

        String patternString = ".*http://.*";

        Pattern pattern = Pattern.compile(patternString);
    }
}
```

You can also use the `Pattern.compile()` method to compile a `Pattern` using special flags. Here is a Java `Pattern.compile()` example using special flags:

```
Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
```

The Java `Pattern` class contains a list of flags (int constants) that you can use to make the `Pattern` matching behave in certain ways. The flag used above makes the pattern matching ignore the case of the text when matching. For more information of the flags you can use with the Java `Pattern` class, see the JavaDoc for `Pattern` .

Pattern.matcher()

Once you have obtained a `Pattern` instance, you can use that to obtain a `Matcher`instance. The `Matcher` instance is used to find matches of the pattern in texts. Here is an example of how to create a `Matcher` instance from a `Pattern` instance:

```
Matcher matcher = pattern.matcher(text);
```

The `Matcher` class has a `matches()` method that tests whether the pattern matches the text. Here is a full example of how to use the `Matcher`:

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

public class PatternMatcherExample {

    public static void main(String[] args) {

        String text    =
            "This is the text to be searched " +
            "for occurrences of the http:// pattern.";

        String patternString = ".*http://.*";

        Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);

        Matcher matcher = pattern.matcher(text);
        boolean matches = matcher.matches();

        System.out.println("matches = " + matches);
    }
}
```

The `Matcher` is very advanced, and allows you access to the matched parts of the text in a variety of ways. Too keep this text short, the `Matcher` covered in more detail in the text about the **Java Matcher class**.

Pattern.split()

The `split()` method in the `Pattern` class can split a text into an array of `String`'s, using the regular expression (the pattern) as delimiter. Here is a Java `Pattern.split()`example:

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

public class PatternSplitExample {

    public static void main(String[] args) {

        String text = "A sep Text sep With sep Many sep Separators";

        String patternString = "sep";
        Pattern pattern = Pattern.compile(patternString);

        String[] split = pattern.split(text);

        System.out.println("split.length = " + split.length);

        for(String element : split){
            System.out.println("element = " + element);
        }
    }
}
```

This `Pattern.split()` example splits the text in the `text` variable into 5 separate strings. Each of these strings are included in the `String` array returned by the `split()` method. The parts of the text that matched as delimiters are not included in the returned String array.

Pattern.pattern()

The `pattern()` method of the `Pattern` class simply returns the pattern string (regular expression) that the `Pattern` instance was compiled from. Here is an example:

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

public class PatternPatternExample {

    public static void main(String[] args) {

        String patternString = "sep";
        Pattern pattern = Pattern.compile(patternString);

        String pattern2 = pattern.pattern();
    }
}
```

In this example the `pattern2` variable will contain the value `sep`, which was the value the `Pattern` instance was compiled from.

Java Regex - Matcher

The Java `Matcher` class (`java.util.regex.Matcher`) is used to search through a text for multiple occurrences of a regular expression. You can also use a `Matcher` to search for the same regular expression in different texts.

The Java `Matcher` class has a lot of useful methods. I will cover the core methods of the Java `Matcher` class in this tutorial. For a full list, see the official JavaDoc for the `Matcher`class.

Java Matcher Example

Here is a quick Java `Matcher` example so you can get an idea of how the `Matcher` class works:

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

public class MatcherExample {

    public static void main(String[] args) {

        String text    =
            "This is the text to be searched " +
            "for occurrences of the http:// pattern.";

        String patternString = ".*http://.*";
        Pattern pattern = Pattern.compile(patternString);

        Matcher matcher = pattern.matcher(text);
        boolean matches = matcher.matches();
    }
}
```

First a `Pattern` instance is created from a regular expression, and from the `Pattern`instance a `Matcher` instance is created. Then the `matches()` method is called on the`Matcher` instance. The `matches()` returns `true` if the regular expression matches the text, and `false` if not.

You can do a whole lot more with the `Matcher` class. The rest is covered throughout the rest of this tutorial. The `Pattern` class is covered separately in my **Java Regex Pattern tutorial**.

Creating a Matcher

Creating a `Matcher` is done via the `matcher()` method in the `Pattern` class. Here is an example:

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

public class CreateMatcherExample {

    public static void main(String[] args) {

        String text    =
            "This is the text to be searched " +
            "for occurrences of the http:// pattern.";

        String patternString = ".*http://.*";

        Pattern pattern = Pattern.compile(patternString);

        Matcher matcher = pattern.matcher(text);
    }
}
```

At the end of this example the `matcher` variable will contain a `Matcher` instance which can be used to match the regular expression used to create it against different text input.

matches()

The `matches()` method in the `Matcher` class matches the regular expression against the whole text passed to the `Pattern.matcher()` method, when the `Matcher` was created. Here is a `Matcher.matches()` example:

```
String patternString = ".*http://.*";
Pattern pattern = Pattern.compile(patternString);
```

```
boolean matches = matcher.matches();
```

If the regular expression matches the whole text, then the `matches()` method returns true. If not, the `matches()` method returns false.

You cannot use the `matches()` method to search for multiple occurrences of a regular expression in a text. For that, you need to use the `find()`, `start()` and `end()` methods.

lookingAt()

The `Matcher` `lookingAt()` method works like the `matches()` method with one major difference. The `lookingAt()` method only matches the regular expression against the beginning of the text, whereas `matches()` matches the regular expression against the whole text. In other words, if the regular expression matches the beginning of a text but not the whole text, `lookingAt()` will return true, whereas `matches()` will return false.

Here is a `Matcher.lookingAt()` example:

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

public class CreateMatcherExample {

    public static void main(String[] args) {

        String text    =
                "This is the text to be searched " +
                "for occurrences of the http:// pattern.";

        String patternString = "This is the";

        Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
        Matcher matcher = pattern.matcher(text);

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

This example matches the regular expression `"this is the"` against both the beginning of the text, and against the whole text. Matching the regular expression against the beginning of the text (`lookingAt()`) will return true.

Matching the regular expression against the whole text (`matches()`) will return false, because the text has more characters than the regular expression. The regular expression says that the text must match the text `"This is the"` exactly, with no extra characters before or after the expression.

find() + start() + end()

The `Matcher` `find()` method searches for occurrences of the regular expressions in the text passed to the `Pattern.matcher(text)` method, when the `Matcher` was created. If multiple matches can be found in the text, the `find()` method will find the first, and then for each subsequent call to `find()` it will move to the next match.

The methods `start()` and `end()` will give the indexes into the text where the found match starts and ends. Actually `end()` returns the index of the character *just after* the end of the matching section. Thus, you can use the return values of `start()` and `end()` inside a `String.substring()` call.

Here is a Java `Matcher` `find()`, `start()` and `end()` example:

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

public class MatcherFindStartEndExample {

    public static void main(String[] args) {

        String text    =
                "This is the text which is to be searched " +
                "for occurrences of the word 'is'.";

        String patternString = "is";

        Pattern pattern = Pattern.compile(patternString);
        Matcher matcher = pattern.matcher(text);

        int count = 0;
        while(matcher.find()) {
            count++;
            System.out.println("found: " + count + " : "
                    + matcher.start() + " - " + matcher.end());
        }
    }
}
```

This example will find the pattern "is" four times in the searched string. The output printed will be this:

```
found: 1 : 2 - 4
found: 2 : 5 - 7
found: 3 : 23 - 25
found: 4 : 70 - 72
```

reset()

The `Matcher` `reset()` method resets the matching state internally in the `Matcher`. In case you have started matching occurrences in a string via the `find()` method, the `Matcher` will internally keep a state about how far it has searched through the input text. By calling `reset()` the matching will start from the beginning of the text again.

There is also a `reset(CharSequence)` method. This method resets the `Matcher`, and makes the `Matcher` search through the `CharSequence` passed as parameter, instead of the `CharSequence` the `Matcher` was originally created with.

group()

Imagine you are searching through a text for URL's, and you would like to extract the found URL's out of the text. Of course you could do this with the `start()` and `end()`methods, but it is easier to do so with the group functions.

Groups are marked with parentheses in the regular expression. For instance:

```
(John)
```

This regular expression matches the text `John`. The parentheses are not part of the text that is matched. The parentheses mark a group. When a match is found in a text, you can get access to the part of the regular expression inside the group.

You access a group using the `group(int groupNo)` method. A regular expression can have more than one group. Each group is thus marked with a separate set of parentheses. To get access to the text that matched the subpart of the expression in a specific group, pass the number of the group to the `group(int groupNo)` method.

The group with number 0 is always the whole regular expression. To get access to a group marked by parentheses you should start with group numbers 1.

Here is a `Matcher` `group()` example:

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

public class MatcherGroupExample {

    public static void main(String[] args) {

        String text    =
                  "John writes about this, and John writes about that," +
                          " and John writes about everything. "
                ;

        String patternString1 = "(John)";

        Pattern pattern = Pattern.compile(patternString1);
        Matcher matcher = pattern.matcher(text);

        while(matcher.find()) {
            System.out.println("found: " + matcher.group(1));
        }
    }
}
```

This example searches the text for occurrences of the word `John`. For each match found, group number 1 is extracted, which is what matched the group marked with parentheses. The output of the example is:

```
found: John
found: John
```

```
found: John
```

Multiple Groups

As mentioned earlier, a regular expression can have multiple groups. Here is a regular expression illustrating that:

```
(John) (.+?)
```

This expression matches the text `"John"` followed by a space, and then one or more characters. You cannot see it in the example above, but there is a space after the last group too.

This expression contains a few characters with special meanings in a regular expression. The . means "any character". The + means "one or more times", and relates to the . (any character, one or more times). The ? means "match as small a number of characters as possible".

Here is a full code example:

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

public class MatcherGroupExample {

    public static void main(String[] args) {

        String text    =
                  "John writes about this, and John Doe writes about that," +
                          " and John Wayne writes about everything."
                ;

        String patternString1 = "(John) (.+?) ";

        Pattern pattern = Pattern.compile(patternString1);
        Matcher matcher = pattern.matcher(text);

        while(matcher.find()) {
            System.out.println("found: " + matcher.group(1) +
                               " "       + matcher.group(2));
        }
    }
}
```

Notice the reference to the two groups, marked in bold. The characters matched by those groups are printed to `System.out`. Here is what the example prints out:

```
found: John writes
found: John Doe
found: John Wayne
```

Groups Inside Groups

It is possible to have groups inside groups in a regular expression. Here is an example:

```
((John) (.+?))
```

Notice how the two groups from the examples earlier are now nested inside a larger group. (again, you cannot see the space at the end of the expression, but it is there).

When groups are nested inside each other, they are numbered based on when the left paranthesis of the group is met. Thus, group 1 is the big group. Group 2 is the group with the expression `John` inside. Group 3 is the group with the expression `.+?` inside. This is important to know when you need to reference the groups via the `groups(int groupNo)`method.

Here is an example that uses the above nested groups:

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

public class MatcherGroupsExample {

    public static void main(String[] args) {

        String text    =
                  "John writes about this, and John Doe writes about that," +
                          " and John Wayne writes about everything."
                ;

        String patternString1 = "((John) (.+?)) ";

        Pattern pattern = Pattern.compile(patternString1);
        Matcher matcher = pattern.matcher(text);

        while(matcher.find()) {
            System.out.println("found: <"  + matcher.group(1) +
                               "> <"       + matcher.group(2) +
                               "> <"       + matcher.group(3) + ">");
        }
    }
}
```

Here is the output from the above example:

```
found: <John writes> <John> <writes>
found: <John Doe> <John> <Doe>
found: <John Wayne> <John> <Wayne>
```

Notice how the value matched by the first group (the outer group) contains the values matched by both of the inner groups.

replaceAll() + replaceFirst()

The `Matcher` `replaceAll()` and `replaceFirst()` methods can be used to replace parts of the string the `Matcher` is searching through. The `replaceAll()` method replaces all matches of the regular expression. The `replaceFirst()` only replaces the first match.

Before any matching is carried out, the `Matcher` is reset, so that matching starts from the beginning of the input text.

Here are two examples:

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

public class MatcherReplaceExample {

    public static void main(String[] args) {

        String text    =
                  "John writes about this, and John Doe writes about that," +
                          " and John Wayne writes about everything."
                ;

        String patternString1 = "((John) (.+?)) ";

        Pattern pattern = Pattern.compile(patternString1);
        Matcher matcher = pattern.matcher(text);

        String replaceAll = matcher.replaceAll("Joe Blocks ");
        System.out.println("replaceAll   = " + replaceAll);

        String replaceFirst = matcher.replaceFirst("Joe Blocks ");
        System.out.println("replaceFirst = " + replaceFirst);
    }
}
```

And here is what the example outputs:

```
replaceAll   = Joe Blocks about this, and Joe Blocks writes about that,
    and Joe Blocks writes about everything.
replaceFirst = Joe Blocks about this, and John Doe writes about that,
    and John Wayne writes about everything.
```

The line breaks and indendation of the following line is not really part of the output. I added them to make the output easier to read.

Notice how the first string printed has all occurrences of `John` with a word after, replaced with the string `Joe Blocks`. The second string only has the first occurrence replaced.

appendReplacement() + appendTail()

The `Matcher` `appendReplacement()` and `appendTail()` methods are used to replace string tokens in an input text, and append the resulting string to a `StringBuffer`.

When you have found a match using the `find()` method, you can call the`appendReplacement()`. Doing so results in the characters from the input text being appended to the `StringBuffer`, and the matched text being replaced. Only the characters starting from then end of the last match, and until just before the matched characters are copied.

The `appendReplacement()` method keeps track of what has been copied into the`StringBuffer`, so you can continue searching for matches using `find()` until no more matches are found in the input text.

Once the last match has been found, a part of the input text will still not have been copied into the `StringBuffer`. This is the characters from the end of the last match and until the end of the input text. By calling `appendTail()` you can append these last characters to the `StringBuffer` too.

Here is an example:

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

public class MatcherReplaceExample {

    public static void main(String[] args) {

        String text    =
                  "John writes about this, and John Doe writes about that," +
                          " and John Wayne writes about everything."
                ;

        String patternString1 = "((John) (.+?)) ";

        Pattern      pattern      = Pattern.compile(patternString1);
        Matcher      matcher      = pattern.matcher(text);
        StringBuffer stringBuffer = new StringBuffer();

        while(matcher.find()){
            matcher.appendReplacement(stringBuffer, "Joe Blocks ");
            System.out.println(stringBuffer.toString());
        }
        matcher.appendTail(stringBuffer);

        System.out.println(stringBuffer.toString());
    }
}
```

Notice how `appendReplacement()` is called inside the `while(matcher.find())` loop, and `appendTail()` is called just after the loop.

The output from this example is:

```
Joe Blocks
Joe Blocks about this, and Joe Blocks
Joe Blocks about this, and Joe Blocks writes about that, and Joe Blocks
Joe Blocks about this, and Joe Blocks writes about that, and Joe Blocks
    writes about everything.
```

The line break in the last line is inserted by me, to make the text more readable. In the real output there would be no line break.

As you can see, the `StringBuffer` is built up by characters and replacements from the input text, one match at a time.

Java Regex - Regular Expression Syntax

Jakob Jenkov
Last update: 2017-11-06

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
```

These three expressions all refer to the uppercase `A` character. The first uses the octal code (`101`) for `A`, the second uses the 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.

Boundary Matchers

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.*?
```

This expression will match the word `John` followed by zero or more characters The `.`means "any character", and the `*` means "zero or more times". The `?` after the `*` makes the `*` a reluctant quantifier.

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).

The `and` operator is implicit. When you write the expression `John`, then it means "`J` and `o`and `h` and `n`".

The `or` operator is explicit, and is written with a `|`. For instance, the expression `John|hurt`will match either the word `John`, or the 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.

Predefined Character Classes

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

Construct	Matches
`XY`	Matches X and Y (X followed by Y).
`X|Y`	Matches X or Y.