Theory: Regexps in Java

14 minutes 0 / 5 problems solved

A **regular expression** is a sequence of characters that specifies a set of strings and that is used to search, edit, and manipulate text. Like most programming languages Java supports regular expressions. We've already learned some basics of the regex language. In this lesson, we'll explore how it is realized in Java.

§1. Simple matching

First of all, we can create a regular expression by means of a `String` . Take a look at the following example:

```
1 String aleRegex = "ale"; // the "ale" regex
```

In Java, `String` data type has built-in support for regular expressions. Strings have a special method called `matches` that takes a regular expression pattern as its argument and checks whether the string matches this pattern. Keep in mind that the method returns `true` only when the *whole* string matches the regexp, otherwise, it returns `false`. The pattern defined by the regex is applied to the text from left to right.

In the example below, we try to match `aleRegex` and different strings:

```
1 "ale".matches(aleRegex);  // true
2
"pale".matches(aleRegex); // false, "pale" string has an additional character
3
"ALE".matches(aleRegex);  // false, uppercase letters don't match lowercase and vi
ce versa
```

You can see that the string `"pale"` is not matched by our regex pattern. The reason is that Java regex implementation checks whether the *whole* string can be fit into the regex pattern, not just some substring. In this regard, Java differs from many other programming languages.

Here is another example. The `helloRegex` pattern has two words separated by a comma and a whitespace character:

```
1 String helloRegex = "Hello, World";
2
3 "Hello, World".matches(helloRegex); // true
4 "Hello, world".matches(helloRegex); // false
5 "Hello,World".matches(helloRegex); // false
```

As is evident from the previous examples, when our regex is just a sequence of simple characters, it can be matched only with the exact same string. There are easier ways to compare strings, though, and we wouldn't really need regular expressions if that was all they could do.

As you remember, the real power of regular expressions lies in special characters that help you define a pattern matching several different strings at once. In this lesson, we will look at two special characters you are already familiar with.

§2. The dot character and the question mark

The dot `.` matches any single character including letters, digits, spaces, and so on. The only character it is unable to match with is the newline character `\n`. The examples below should look familiar to you:

Current topic:

Regexps in Java

Topic depends on:

Regexps basics

Boolean and logical operations — Stage 2

String — Stage 2

Topic is required for:

Sets, ranges, alternations

Table of contents:

↑ Regexps in Java

§1. Simple matching

§2. The dot character and the question mark

§3. The tricky escape character

§4. Conclusion

Feedback & Comments

```
String learnRegex = "Learn.Regex";

"Learn Regex".matches(learnRegex); // true
"Learn.Regex".matches(learnRegex); // true
"LearnRegex".matches(learnRegex); // false
"Learn, Regex".matches(learnRegex); // false
"Learn\nRegex".matches(learnRegex); // false
```

As you remember, the question mark `?` is a special character that means "the preceding character or nothing". Words with slightly different spelling in American and British English serve as a traditional example of this character's application:

```
String pattern = "behaviou?r";

"behaviour".matches(pattern); // true
"behavior".matches(pattern);  // true
```

Now let's combine the dot character `.` and the question mark `?` in one regex pattern. This combination basically means "there can be any single character or no character at all":

```
String pattern = "..?";

"I".matches(pattern);  // true
"am".matches(pattern); // true
"".matches(pattern);   // false
```

Sometimes we want to use the dot character or the question mark, though. Let's see what we should do about that.

§3. The tricky escape character

Right now you're probably wondering what should we do if we want to use the dot `.` or the `?` as a regular punctuation mark and not as a special symbol within the regex pattern?

Well, in this case, we should protect our special symbol by putting the backslash `\` before it. The backslash is called an escape character because it helps symbols to "escape" their working duties. Note that when you want to use the backlash `\` itself in its literal meaning, you need to escape it as well! This way, a double backslash `\\` in your regex means a single backslash in the matching string.

However, it gets more complicated when you implement such patterns in your Java program. The backslash `\` works as an escape character not only for regular expressions but for `String` literal as well. So, in fact, we have to use an additional backslash to escape the one we need in the regular expression, just like this:

```
String endRegex = "The End\\.";

"The End.".matches(endRegex); // true
"The End?".matches(endRegex); // false
```

For instance, the regular expression for any five-character sequence that ends with a dot looks like this:

```
String pattern = ".....\\.";

"a1b2c.".matches(pattern); // true
"Wrong.".matches(pattern); // true
"Hello!".matches(pattern); // false
```

§4. Conclusion

As you can see, regular expressions are a powerful tool for processing strings in Java. They allow us to define common patterns by using regular characters and characters with special meaning, and then check whether strings match these patterns. The key points of this lesson are:

- `matches` string method is used to compare a regex pattern with a string;
- it returns `true` only when the *whole* string matches the regexp;
- when using regexps in Java, we should add extra backslash for escaping symbols, which results in using a double backslash instead of just one.

In the following topics, we will see what other possibilities the Java regexes provide: more special characters, other built-in string methods that take regex patterns, and some advanced classes to process strings.

Report a typo

315 users liked this theory. **10** didn't like it. **What about you?**

Start practicing

Comments (11) Hints (0) Useful links (1) Show discussion