Theory: Patterns and Matcher

43 minutes 0 / 5 problems solved

The Java Class Library has two special classes possessing advanced features for work with regular expressions: `java.util.regex.Pattern` and `java.util.regex.Matcher`. A `Matcher` object provides us with many useful methods to handle regexes, while a `Pattern` object represents a regular expression itself.

§1. Matching a regex

Suppose we have a text stored in a string variable:

```
1 String text = "We use Java to write modern applications";
```

We want to use a regular expression to check whether the text contains substrings "Java" or "java". We can carry this out in three simple steps by means of `Pattern` and `Matcher` classes.

1. Create an object of the `Pattern` class by passing a regex string to the `compile` method:

```
1 Pattern pattern = Pattern.compile(".*[Jj]ava.*"); // regex to match "java" or "Java" in a text
```

2. Create a `Matcher` by invoking the `matcher` method of the `Pattern` and creating an object for the given string:

```
1 Matcher matcher = pattern.matcher(text); // it will match the passed text
```

3. Invoke the `matches` method of the matcher to match the string:

```
1 boolean matches = matcher.matches(); // true
```

The method `matches` of a `Matcher` works exactly the same way as the method `matches` of the `String`, with which we are already familiar.

§2. Advantages of Pattern and Matcher classes

For the moment it may seem that there's no point in `Pattern` and `Matcher` since we already have simple string regex representation. However, there are two main reasons to pay attention to these classes:

- **Performance.** Actually, the `matches` method of the `String` internally invokes the `matches` method of the `Matcher`, but it also invokes `Pattern.compile(...)` every time it is executed. That's not efficient. If the same pattern is used multiple times, compiling it once will be more reasonable.
- **Rich API.** The `Matcher` class has more to offer than a single `matches` method: there are a lot of useful methods to process strings and a `Pattern` provides us with the opportunity to configure it in detail, for example, enable case-insensitive matching.

So, if you plan on reusing your regex several times and/or need more elaborate methods for text and pattern comparison, it is more preferable to use `Pattern` and `Matcher` rather than `String`.

§3. Patterns and Modes

As you know, a `Pattern` is used to create an object of `Matcher`. If we aren't going to reuse our regex, though, we can simply invoke the `matches` method of the `Pattern` class in a single line.

Current topic:

Patterns and Matcher

Topic depends on:

Regexes in programs

Objects Stage 3

Topic is required for:

Replacing characters

Match results

Table of contents:

↑ Patterns and Matcher

§1. Matching a regex

§2. Advantages of Pattern and Matcher classes

§3. Patterns and Modes

```
Pattern.matches(".*
[Jj]ava.*", "We use Java to write modern applications"); // true
```

It is similar to invoking the `matches` method of a `String` but has the same performance problem.

Consider the previous example again. It cannot match words like "JAVA" because it does not ignore the case, as all regular expressions do by default. Fortunately, there is a special mode `Pattern.CASE_INSENSITIVE` that can be set during the compilation of the `Pattern`. It allows your regex to match strings without taking the case into account.

```
Pattern pattern = Pattern.compile(".*java.*", Pattern.CASE_INSENSITIVE);

String text = "We use Java to write modern applications";

Matcher matcher = pattern.matcher(text);

System.out.println(matcher.matches()); // true
```

Another mode you may want to remember is `Pattern.DOTALL` that makes the dot metacharacter `.` match all characters, including the line break `\n`.

> Case-insensitive mode is available even without the `Matcher`. You just need to add `(?i)` at the beginning of your regex. To make the dot character match the newline character, add `(?s)`. You can enable both modes by writing `(?is)`.

Take a look at how this works:

```
Pattern.matches("(?is).*java.*", "\n\nJAVA\n\n"); // true
```

There are also other modes, but we will not consider them here. See documentation for details.

§4. The matches and find methods

An instance of `Matcher` provides us with curious methods for pattern/string matching. In this lesson, we will consider only one of them.

Just as the `matches` method of the `String,` method `matches` of the `Matcher` returns `true` only when the pattern matches the whole string, otherwise, it returns `false`. That's not very convenient in some situations, right? For example, if we want to check whether there is a particular substring somewhere in our text, we have to add `.*` at the beginning and at the end of the pattern.

Thanks to the `Matcher` , we can also apply the `find` method. It is similar to the `matches` , but instead of checking the match with the whole string, it tries to find a substring that matches the pattern. Look at the following example to understand the difference between these methods:

```
String text = "Regex is a powerful tool for programmers";

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


System.out.println(matcher.matches()); // false, the whole string does not match t
he pattern

System.out.println(matcher.find()); // true, there is a substring that matches the
 pattern
```

Remember the boundary characters we've learned before? They can be applied to modify the behavior of the `find` method to make it work somewhat similar to the `matches` method. To make sure that the `find` method will match a substring located at the beginning of the string, we can add the

§4. The matches and find methods

§5. Conclusion

Feedback & Comments

hat character `^` at the start of the regex. To make it match a substring at the end of the string, we can add the dollar character `$` at the end of the regex. By combining these symbols, we make out of `find` a copy of `matches`:

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

System.out.println(matcher.matches()); // false
System.out.println(matcher.find());   // false
```

By default, both methods `matches` and `find` work with the whole string. It is possible, though, to narrow down their scope by invoking the `range` method that allows us to specify the first (inclusive) and the last (exclusive) indices of the substring that we want our methods to consider.

§5. Conclusion

There are two ways to process regexes: by calling the method of the `String`, and by using `Pattern` and `Matcher` classes. The second way is more efficient, and it also provides a set of useful methods and configurations for string processing. There are two main methods, `matches` and `find` , with a key difference. The `matches` method matches the whole string, while the `find` method looks for a substring matching the regex.

Report a typo

182 users liked this theory. **17** didn't like it. **What about you?**

Start practicing

Comments (10) Hints (0) Useful links (0) Show discussion