Patterns and Matcher

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.

Matching a regex

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 with thePattern and Matcher classes.

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

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:

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

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

Advantages of Pattern and Matcher 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, enabling 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 preferable to use Pattern and Matcher rather than String.

Patterns and Modes

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.

```
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.

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.

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

The matches and find methods

Just as the matches method of the String, the 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.

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.

```
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 the 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 similarly to thematches method. To make sure that the find method will match a substring located at the beginning of the string, we can add the 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 region method that allows us to specify the first (inclusive) and the last (exclusive) indices of the substring that we want our methods to consider.

```
String text = "Regex is a powerful tool for programmers";
Matcher matcher = Pattern.compile("tool").matcher(text);

matcher.region(10, 20); // start index = 10, end index = 20
System.out.println(matcher.find()); // false
matcher.region(20, 30); // start index = 20, end index = 30
System.out.println(matcher.find()); // true
```