Assignment1 notes

Regular expression in Java

Reference:

> Java 正则表达式 | 菜鸟教程 (runoob.com)
>
> Java 之正则表达式语法及常用正则表达式汇总 - 知乎 (zhihu.com)
>
> javascript - Difference between ?:, ?! and ?= - Stack Overflow
>
> java.util.regex (Java SE 17 & JDK 17) (oracle.com)

1. what is regular expression:

   a pattern string that can match a group of strings with common characteristics.

2. what can we do with a pattern string

   1. find whether it matches a text
   2. find whether it's in a text
   3. replace substrings in a string with new Strings
   4. use it as a separator

3. Classes in java.util.regex

   1. `Pattern`

      The pattern object is a compiled representation of a regular expression.

      The Pattern class has no **public constructor**.

      To create a Pattern object, you must first call its `public static compile()` method, which returns a Pattern object. This method accepts a regular expression as its first parameter.

      ```
      Pattern p = Pattern.compile("1[0-9]{7}");
      ```

   2. `Matcher`:
      The Matcher object is an engine that interprets and matches input strings.

      Like the Pattern class, Matcher has **no public constructor**.

      You need to call the `public static matcher` method of a `Pattern`object to get a **Matcher** object.

      ```
      Pattern p = Pattern.compile("1[0-9]{7}");
      Matcher m = p.matcher("12345678");
      ```

      Some commonly used methods are listed here

      - matching methods

method	description

method	description
`public boolean lookingAt()`	try to match the head of the string
`public boolean matches()`	try to match the entire string
`public boolean find()`	try to find the next matched substring in a string

- replace methods

method	description
`public String replaceFirst (String replacement)`	Replaces the **first** subsequence of the input sequence that matches the pattern with the given replacement string.
`public String replaceFirst (Function<MatchResult, String> replacer)`	Replaces the **first** subsequence of the input sequence that matches the pattern with the result of **applying the given replacer function**
`public String replaceAll (String replacement)`	Replaces **every** subsequence of the input sequence that matches the pattern with the given replacement string.
`public String replaceAll (Function<MatchResult, String> replacer)`	Replaces **every** subsequence of the input sequence that matches the pattern with the result of **applying the given replacer function**

For more you should refer to:

> java.util.regex (Java SE 17 & JDK 17) (oracle.com)

3. `PatternSyntaxException`:
   PatternSyntaxException is an optional exception class that indicates a syntax error in a regular expression pattern.

   methods

index	method signature
1	**public String getDescription()**
2	**public int getIndex()**
3	**public String getPattern()**
4	**public String getMessage()** Returns a multiline string containing a description of the syntax error and its index, the erroneous regular expression pattern, and a visual indication of the error's index in the pattern.

4. Some Methods in String also support regex
    - if your task is simple enough, no need to use `Pattern`, `Matcher`, just use method `String`, because it support the most commonly used methods.

```
// matches
s1.matches(regex);
// split
/* split machanism:
continuely find() and use it as a seperator
*/
String [] strs = s1.split(regex);
// replaceAll
s1.replaceAll(regex, newString)
// replaceAll
s1.replaceFirst(regex, newString)
```

5. syntax: what does a regular expression match
    1. common character:

       Letters, numbers, Chinese characters, underscores, and punctuation marks without special definitions are all "common characters". Ordinary characters in the expression, when matching a string, match the same character.

char	description
`[]`	match **one** character that is in the `[]`. For example, `[ABC]` match **A**, **B** or **C**
`[^]`	match **one** character that isn't in the `[]`. For example, `[^AB]` don't match **A** and **B**
`[ - ]`	match **one** character that is in the **interval**. For example, `[0-9]` match 0-9
`.`	don't match `\n` or `\r`, i.e. `[^\n\r]`
`\\d`	0-9
`\\w`	A-Z, a-z, 0-9, _
`\\s`	space, tab, form feed, blank character

    2. escape character:

char	description
`\r, \n`	enter, newline
`\t`	tab
`\\`	\

char	description
`\^`	^
`\$`	$
`\.`	.

3. matched times

expression	description
{n}	the **character** or **subexpression** should repeat n times For example, "a{5}" equals to "aaaaa"
{m,n}	the **character** or **subexpression** should repeat at least m times, and at most n times. For example, "a{2, 3}" matched "aa" or "aaa"
{m,}	the **character** or **subexpression** should repeat at least m times. For example, "a{2}" matched "aaaaaa"
?	matched 0 or 1 time, i.e. {0,1}
+	matched at least 1 time, i.e.{1,}
*	matched at least 0 time, i.e.{0,}

4. operators

expression	description
`|`	means "or"
`()`	generate a subexpression

```java
System.out.println("zoo".matches("(f|o)(oo)*")); // true

// to match a ( or )
System.out.println("zoo()".matches("(f|o)(oo)*[(][)]")); // true
```

5. special marks

expression	description
^	marks the begining of a string
$	marks the end of a string

```
System.out.println(Pattern.compile("oo").matcher("food").find()); // true

System.out.println(Pattern.compile("^fo").matcher("food").find()); // true
System.out.println(Pattern.compile("^oo").matcher("food").find()); // false

System.out.println(Pattern.compile("od&").matcher("food").find());  // true
System.out.println(Pattern.compile("oo&").matcher("food").find());  // false
```

6. matching and capture

   It seems the same as before in matching. However, in replacement and seperations, distinction between matching and capture are significant.

expression	description
(*pattern*)	**Matches** *pattern* and **captures** subexpressions of that match. **Captured matches** can be retrieved from the resulting "match" collection using the **0…9** indices.
(?:*pattern*)	**Match** *pattern* **without capturing** subexpressions of that match. Such pattern in the text will be counted in the **matched part**.
(?=*pattern*)	matches the string at the beginning of the string matching pattern. It is a non-capturing match. The lookahead **does not occupy characters**, that is, after a match occurs, the search for the next match **follows the previous match, not after the characters that make up the lookahead.**
(?!*pattern*)	matches a search string that is **not** at the start of a string matching *pattern*. It is a non-capturing match. The lookahead **does not occupy characters**, that is, after a match occurs, the search for the next match **follows the previous match, not after the characters that make up the lookahead.**

```
// see difference between ?: and ?=
String s = "Java, C, C++, Python";
String [] sarr1 = s.split(", (?:[\\w+]*, [\\w+]*)");
String [] sarr2 = s.split(", (?=[\\w+]*, [\\w+]*)");
for(String s0: sarr1){
    System.out.println(s0);
}
System.out.println();
for(String s0: sarr2){
    System.out.println(s0);
}
```

```
/* output:
Java
, Python

Java
C
C++, Python
*/
```

Another example can be seen in:

> javascript - Difference between ?:, ?! and ?= - Stack Overflow