# Regular Expressions

**Fundamentals** A **regular expression** (or regex) is a character or a sequence of characters that represent a string or multiple strings.

Regular expressions are part of the **java.util.regex** package; this package has classes to help form and use regular expressions. Regular expressions allow for quicker, easier searching, parsing, and replacing of characters in a string.

Regular expressions are used to ensure that strings contain specific contents and are often used to check correct email format (@ sign is included) on form validation.

The String class contains a method named **matches(String regex)** that returns true if a string matches the given regular expression. This is similar to the String method **equals(String str)**. The difference is that comparing a string to a regular expression allows variability.

For example, you are to write code that returns true if the String named

animal is **"**cat**" or "**dog**"**, and returns false otherwise.

The **equals()** method compares one (1) string to another by comparing a single argument.

# Example:

**if(animal.equals("cat")) return true;**

**else if(animal.equals("dog")) return true;**

**return false;**

The **matches()** method allows you to determine what the string should match against and allows you to specify multiple values. This method allows much more flexibility in your code. The equivalent of the above sample code using regular expressions is:

# return animal.matches("cat|dog");

The regular expression symbol **|** allows for the method matches to check if

animal is equal to "cat" or "dog" and return true accordingly.

Square brackets are used in regular expressions to allow for character variability. For example, you wanted to return true if animal is equal to "dog" or "Dog", but not "dOg". Using equalsIgnoreCase would not work, and using equals would take multiple lines. If you use regular expression, this task can be done in a single statement:

# return animal.matches("[Cc]at|[Dd]og");

Square brackets are not restricted to two-character options. They can be combined with a hyphen to include any range of characters.

For example, you are to write code to create a rhyming game. You want to check if a string word rhymes with **"**cat**"**. The code shall return true if a word begins with any lowercase letter and ends in **"**at**"**. Your first attempt at coding may look like this:

**if(word.length()==3) if(word.substring(1,3).equals("at"))**

**return true; return false;**

Regular expressions make it much easier to solve similar problems. A shorter, more generic way to complete the same task is to use square brackets and a hyphen (regular expression), as shown below.

# return word.matches("[a-z]at");

To include uppercase characters, write:

# return word.matches("[a-zA-Z](at|AT)");

This allows specific characters to be upper or lowercase. Using the range a-Z or A-z will not work correctly.

Everything inside of the square brackets defines exactly how the first character can be represented. For example:

# return word.matches("[ 0-9a-zA-Z]bcde");

The regular expression allows the first character to be a space, any number, or a lower/upper case character. The rest of the word must be **"bcde"** and is case-sensitive.

The **dot** (.) is the wildcard operator that represents any single character in regular expressions. For example, you are to write code that checks if a String element consists of a number followed by any other single character. Your statement would look like this:

# return element.matches("[0-9].");

**Repetition Operators** A **repetition operator** is any symbol in regular expressions that indicates the number of times a specified character appears in a matching string.

|  |  |  |  |
| --- | --- | --- | --- |
| **Repetition Operator** | **Definition** | **Sample Statement** | **Code Meaning** |
| \* | 0 or more occurrences | return str.matches("A\*"); | Returns true if str  consists of zero or more A's but no  other letter |
| ? | 0 or 1 occurrence | return str.matches("A?"); | Returns true if str  is "" or "A" |
| + | 1 or more occurrences | return str.matches("A+"); | Returns true if str  is 1 or more A's in a sequence |
| {x} | x occurrences | return str.matches("A{7}"); | Returns true if str  is a sequence of 7 A's |
| {x,y} | Between x & y occurrences | return str.matches("A{7,9}"); | Returns true if str  is a sequence of 7, 8, or 9 A's |
| {x,} | x or more occurrences | return str.matches("A{5,}"); | Returns true if str  is a sequence of 5 or more A's. |

Regular expressions allow for very powerful validation of strings without having to write much code. For example:

# return str.matches(".\*");

The dot represents any character, while the asterisk represents any number of occurrences of the character preceding it. Hence, "**.\***" means any number of any characters in a sequence will return true.

All symbols of regular expressions may be combined with each other. For example:

# return str.matches(".?[0-5]{10}");

**Pattern and Matcher** The **Pattern** class in the java.util.regex package stores the format of a regular expression.

For example, the statement below initializes a pattern of characters as defined by the regular expression "[A-F]{5,}.\*"

# Pattern p = Pattern.compile("[A-F]{5,}.\*");

The **compile** method compiles the given regular expression into a pattern. A compiled regex pattern can speed up your program when the pattern is used frequently.

The **Matcher** class in the java.util.regex package stores a possible match between a pattern and a string.

A matcher is initialized as follows:

# Matcher match = patternName.matcher(stringName);

The **matcher** method returns a Matcher object. Use the **matches()** method to check if the regular expression given in the Pattern declaration matches a given string. For example:

# return match.matches();

The following sample code checks if the String str conforms to the defined

Pattern p. The Matcher named match compares the string and pattern.

**Pattern p = Pattern.compile("[A-F]{5,}.\*"); String str = "AAAAAhhh";**

**Matcher match = p.matcher(str); System.out.println(match.matches()); //returns true**

**RegEx Operations** Matcher's **find()** method scans the input sequence looking for the next subsequence that matches the pattern. It will return true if the defined pattern exists as a substring of the string of the matcher.

**Pattern p = Pattern.compile("[0-9]"); String str = "Passw0rd";**

**Matcher match = p.matcher(str); System.out.println(match.find()); //returns true**

The **split()** method splits the string around matches of the given regular expression. It is used to specify where the operator wishes to split the string.

**String phone = "991-978-6452"; String[] tokens = str.split("-"); for (String t : tokens)**

**System.out.println(t);**

**/\*Output:**

**991**

**978**

**6452**

**\*/**

The **replaceAll()** method is used to replace all the occurrences of the defined regular expression found in the string with another string. The sample code below uses a regular expression to remove multiple spaces from a string and replaces them with a substring that consists of a single space.

**String str = "Help on!";**

**me**

**I have**

**no idea what's going**

**str = str. replaceAll(" {2,}", " "); System.out.println(str);**

**//Output: Help me I have no idea what's going on!**

**References:**

Farrell, J. (2019). *Java programming.* Cengage Learning, Inc.

Oracle Academy (n.d.). Citing sources. Retrieved from https://academy.oracle.com/ Oracle Docs (n.d.). Citing sources. Retrieved from

https://docs.oracle.com/javase/tutorial/java/javaOO/index.html