# RegEx Guide

Spring 2020

## Contents:


What Exactly is RegEx?
- Example: Writing A RegEx

Characters and Quantifiers
- Characters
- Quantifiers
- Escaping Characters

How to RegEx (Java Edition)
- Splitting a String
- Matching a String
- Replacing a Substring
- Patterns and Matchers

How to RegEx (Scala Edition)
- Extracting Matches in Scala


# What Exactly is RegEx?

RegEx is short for Regular Expression. A RegEx is a sequence of characters that defines a search pattern. They allow you to create a set of rules for the kind of String you want, and then search through a larger body of text in order to find Strings that fit these rules.

## Example: Writing A RegEx

At the most basic level, a RegEx can allow you to find a specific String within a larger body of text. For example, if we were just looking for every instance of the word “cat” in a larger document, we would use the RegEx “cat”. However, this won’t get us very far practically. We can do much more than this.

Let’s say instead, we’re looking for every 3 letter word in a body of text that ends in “at” (e.g. “cat”, “bat”, “sat”, etc). For this, we can use “[a-z]at”. The brackets create a set of characters to search for, and the a-z gives a range of every character between a and z in the ASCII character list.

Now instead, suppose we want every single word ending in “at”, regardless of length (e.g. “at”, “bat”, “seat”, etc). For this, we can use a quantifier. The * quantifier says that there can be any number, including 0, of the character proceeding it in the string it finds. This means “[a-z]*at” will do what we want; find every String consisting of some number of letters followed by “at”. (If we want to exclude “at” and only find words with 3 or more letters, we could instead use “[a-z]+at”, as the + quantifier requires there be one or more a-z characters.)

This RegEx still has some issues though. For example, if we used it to find all of the words ending in “at” in the string “The cat was late,” it would match “cat”, but it would also match “lat”. We need some way to specify that the “at” actually be at the end of the word. For this, we can use a “word boundary,” which requires that there be “word characters” (letters, numbers or underscores) on only one side of the boundary. This is represented as “\b” in the RegEx syntax. So our new RegEx, “[a-z]*at\b”, will match “at”, “cat”, “seat”, and “great”, but not “ate”, “late”, “bat9”, or “at_home”. If we want to avoid matches like “seat” or “great”, we could add a word boundary to the beginning as well, for a final RegEx of “\b[a-z]*at\b”.

# Characters and Quantifiers

## Characters

- \w : Matches any word character (letters, digits or underscores)
- \d: Matches any digit
- \s: Matches any whitespace character (space, tab, newline)
- \W: Matches any non-word character (NOT a letter, digit, or underscore)
- \D: Matches any non-digit character
- \S: Matches any non-whitespace character
- \b: Matches a word boundary (there must be word characters on only one side of the boundary)
- . : (A period) Matches any one character except a newline.
- [xyz]: Matches any character within the brackets (ie. x or y or z)
- [x-z]: Matches any character in the range between x and z on the ASCII character list
- [^x]: Matches any character that is not x
    - Can be combined with other techniques. (ie. [^b-x] matches any character not in the range b-x)
- (abc): Matches the string inside the parentheses (ie. “abc”)
- ^: If at the beginning of the RegEx, requires that the pattern be found at the start of the String being searched.
- $: If at the end of the RegEx, requires that the pattern end at the end of the String being searched.

## Quantifiers

- +: One or more of the preceding character
- {x}: Exactly x of the preceding character
- {x, y}: Between x and y of the preceding character
- {x, }: x or more of the preceding character
- *: 0 or more of the preceding character
- ?: 0 or 1 of the preceding character

If no quantifier is present on a character or set of characters, the regex will look for examples with exactly one of the character.

## Escaping Characters

A number of characters have special meanings in regular expressions. For example, "." is used as a wildcard, matching any one character besides a newline. But what if we're actually looking for every period in a document? For this, "\." will find every character that's actually a period. This is known as "escaping" the special character. This will work for any character that has a special meaning in regular expressions (e.g. ()[]*+?\^$).

# How to RegEx (Java Edition)

In Java, how you instantiate a regex depends on what you want to do with it. If you are just trying to see if an input matches a regex or if you just want to split on a regex, you can treat it as an ordinary String. If you want to extract matches of a regex or do anything more complicated, you have to instantiate a Pattern and a Matcher to do that for you.

Note: In Java, if you want to use a backslash, remember to escape it. For example, \s+ needs to be written \\s+.

## Splitting a String

If you want to split a String on a regex, you can simply use the String class's `split` method. For this, you do not need to compile a Pattern.

Given a String `str` and a regex `r`, the syntax for splitting `str` is:

```
str.split(r);
```

This method outputs an array of Strings representing the original String split up by the regex.

Ex: To split on one or more whitespace:

```
String a = "RegExes are         fun";
String[] splitStr = a.split("\\s+");
```

At the end `splitStr` will look like this: {“RegExes”, “are”, “fun”}

## Matching a String

To determine whether a String matches a regex, you can use the `matches` method from the String class. For this, you also do not need to compile a Pattern.

Given a String `str` and a regex `r`, the syntax for matching is:
`str.matches(r);`
This method outputs a boolean - true if it matches, false if not.

Ex1:
```
String a = “hello”;
String regex1 = “[a-z]+”;
a.matches(regex1);
```

This evaluates to true.

Ex2:
```
String b = “18”;
String regex2 = “\\d”;
b.matches(regex2);
```

This evaluates to false because the String contains two digits, not one.

## Replacing a Substring

To replace instances of a substring within a larger String with a different String, you can use the `replaceAll` or `replaceFirst` methods. Both take in a regex and a replacement String.

Given a regex and a replacement String, you can use:

- `String.replaceAll(regex, replacement)`
  to replace all instances that match the regex
- `String.replaceFirst(regex, replacement)`
  to replace the first instance that matches the regex

Ex: Replace the colons with spaces
```
String toReplace = “hours:minutes:seconds”;
toReplace.replaceAll(“:”, “ ”);
```

`toReplace` now is `“hours minutes seconds”`.

## Patterns and Matchers

Unfortunately, the String class does not have a method to allow you to extract matches from a regex. For that, you need to instantiate a Pattern and a Matcher. The Pattern class `(java.util.Regex.Pattern)` essentially takes in a regex and compiles it for use.

To instantiate a Pattern, use the `Pattern.compile` method:

```
Pattern p = Pattern.compile(“\\d+”);
```

This Pattern should match instances of digits. If you want to use a different regex, replace the `“\\d+”` with your own regex!

Now, you can instantiate a matcher to extract pieces of a String that match the Pattern. To do this:

```
Matcher m = p.matcher(“cs18isgr8”);
```

The matcher is instantiated using a pattern to match on a String (in this case “cs18isgr8”).

Afterwards, to extract matches, use the `Matcher.find()` command, which attempts to find the next subsequence of the input sequence that matches the pattern, and outputs a boolean if successful. To access the matches, use `Matcher.group()`. If you instead want to extract a particular match (ie. the first), you can also use `Matcher.group(0)`.

For example, here is a pattern and a matcher used to extract matching Strings:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Regex1 {
    public static void main(String[]args) {
        Pattern p = Pattern.compile("\\d+");
        Matcher m = p.matcher("hello1234goodjob789you2345");
        while(m.find()) {
            System.out.println(m.group());
        }
    }
}
```

This will print:

```
1234
```

```
789
2345
```

Note: All of the earlier methods (splitting, replacing, matching, etc) can also be done using Patterns and Matchers. To find out how, visit the Patterns and Matchers documentation at:
https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html
And
https://docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html

# How to RegEx (Scala Edition)

In Scala (which you will be doing Search in), in order to create and use a regex, you have to instantiate one from the regex class (`scala.util.matching.Regex`). There are a few ways to do this.

```
val a = new Regex("\\s+");
val b = raw"\s+".r;
val c = new Regex("""\s+""");
```

All of these regexes match on whitespace. However, you might have noticed that in the second and third version, we use \s+, whereas in the first, we use \\s+. This is because the raw and triple quote ways of instantiating a regex do not require you to escape a backslash. Feel free to use any of these methods to instantiate your regexes in this class.

A lot of the regex functionality for Scala is the same as for Java. For example, splitting a String and replacing a substring with a new String is still located in the String class and can be called with the same syntax as in Java.

However, in order to extract the matches, Scala has slightly different syntax.

## Extracting Matches in Scala

To find and get the matches of a regex in Scala, use `findFirstMatchIn` and `findAllMatchIn`.

`findFirstMatchIn` takes in the String to match the regex on, and returns an Option.

If we have

```
val reg = new Regex("""\d+""");
findFirstMatchIn("as9");
```

This would return `Some("9")`.

We need `findAllMatchesIn` to get all of the matches for a regex and a specific String. `findAllMatchesIn` takes in the same arguments as `findFirstMatchIn`, but it returns an Iterator over the matches. You can then use the Iterator to either get a List or use the `hasNext` and `next` commands to directly iterate over the matches.