Regular expression

1.1 Regex Syntax Summary

- **Character**: All characters, except those having special meaning in regex, matches themselves. E.g., the regex `x` matches substring `"x"`; `9` matches `"9"`; `=` matches `"="`, `@` matches `"@"`.
- **Special Regex Characters**: `.`, `+`, `*`, `?`, `^`, `$`, `(`, `)`, `[`, `]`, `{`, `}`, `|`
- **Escape Sequences (`\char`)**:
  - To match a character having special meaning in regex, you need to use a escape sequence prefix with a backslash (\). E.g., `\.` matches `"."`; and `\+` matches `"+"`
  - You also need to use `\\` matches `"\"` (back-slash).
  - Regex recognizes common escape sequences such as `\n` for newline, `\t` for tab, `\r` for carriage-return, `\nnn` for a up to 3-digit octal number, `\xhh` for a two-digit hex code, `\uhhhh` for a 4-digit Unicode, `\uhhhhhhhh` for a 8-digit Unicode.
- **A Sequence of Characters (or String)**: Match via combining sub-expressions. E.g., the regex `Saturday` matches `"Saturday"`. The matching, by default, is case-sensitive, but can be set to case-insensitive via modifier.
- **OR Operator (`|`)**: E.g., the regex `four|4` accepts strings `"four"` or `"4"`.

- **Character class (or Bracket List)**:
  - `[...]`: ANY ONE of the character, e.g., `[aeiou]` matches `"a"`, `"e"`, `"i"`, `"o"` or `"u"`.
  - `[.-.]` (Range Expression): ANY ONE of the character in the *range*, e.g., `[0-9]` matches any digit.
  - `[^...]`: NOT ONE of the character, e.g., `[^0-9]` matches any non-digit.
  - Only these four characters require escape sequence inside the bracket list: `^`, `-`, `]`, `\`.
- **Occurrence Indicators (or Repetition Operators)**:
  - `+`: one or more (`1+`), e.g., `[0-9]+` matches one or more digits such as `'123'`, `'000'`.
  - `*`: zero or more (`0+`), e.g., `[0-9]*` matches zero or more digits. It accepts all those in `[0-9]+` plus the empty string.
  - `?`: zero or one (optional), e.g., `[+-]?` matches an optional `"+"`, `"-"`, or an empty string.
  - `{m,n}`: `m` to `n` (both inclusive)
  - `{m}`: exactly `m` times
  - `{m,}`: `m` or more (`m+`)
- **Metacharacters**: matches a character
  - `.` (dot): ANY ONE character except newline. Same as `[^\n]`
  - `\d`, `\D`: ANY ONE digit/non-digit character. Digits are `[0-9]`
  - `\w`, `\W`: ANY ONE word/non-word character. For ASCII, word characters are `[a-zA-Z0-9_]`
  - `\s`, `\S`: ANY ONE space/non-space character. For ASCII, whitespace characters are `[ \n\r\t\f]`
- **Position Anchors**: does not match character, but position such as start-of-line or end-of-word.
  - `^`, `$`: start-of-line and end-of-line respectively. E.g., `^[0-9]$` matches a numeric string.
  - `\b`: boundary of word, i.e., start-of-word or end-of-word. E.g., `\bcat\b` matches the word `"cat"` in the input string.
  - `\B`: Inverse of `\b`, i.e., non-start-of-word or non-end-of-word.
  - `\<`, `\>`: start-of-word and end-of-word respectively, similar to `\b`. E.g., `\<cat\>` matches the word `"cat"` in the input string.
  - `\A`, `\Z`: start-of-input and end-of-input respectively.
- **Parenthesized Back References**:
  - Use parentheses `( )` to create a back reference.
  - Use `$1`, `$2`, ... (Java, Perl, JavaScript) or `\1`, `\2`, ... (Python) to retreive the back references in sequential order.
- **Laziness (Curb Greediness for Repetition Operators)**: `*?`, `+?`, `??`, `{m,n}?`, `{m,}?`

1.2 Example: Numbers `[0-9]+` or `\d+`

1. A regex (*regular expression*) consists of a sequence of *sub-expressions*. In this example, `[0-9]` and `+`.
2. The `[...]`, known as *character class* (or *bracket list*), encloses a list of characters. It matches any SINGLE character in the list. In this example, `[0-9]` matches any SINGLE character between 0 and 9 (i.e., a digit), where dash (`-`) denotes the *range*.
3. The `+`, known as *occurrence indicator* (or *repetition operator*), indicates one or more occurrences (`1+`) of the previous sub-expression. In this case, `[0-9]+` matches one or more digits.
4. A regex may match a portion of the input (i.e., substring) or the entire input. In fact, it could match zero or more substrings of the input (with global modifier).
5. This regex matches any numeric substring (of digits 0 to 9) of the input. For examples,
   a. If the input is `"abc123xyz"`, it matches substring `"123"`.
   b. If the input is `"abcxyz"`, it matches nothing.
   c. If the input is `"abc00123xyz456_0"`, it matches substrings `"00123"`, `"456"` and `"0"` (three matches).

   Take note that this regex matches number with leading zeros, such as `"000"`, `"0123"` and `"0001"`, which may not be desirable.
6. You can also write `\d+`, where `\d` is known as a *metacharacter* that matches any digit (same as `[0-9]`). There are more than one ways to write a regex! Take note that many programming languages (C, Java, JavaScript, Python) use backslash `\` as the prefix for escape sequences (e.g., `\n` for newline), and you need to write `"\\d+"` instead.

Code Example in Java

See "Regular Expressions (Regex) in Java" for full coverage.

Java supports Regex in package `java.util.regex`.

```
1  import java.util.regex.Pattern;
2  import java.util.regex.Matcher;
3
4  public class TestRegexNumbers {
5     public static void main(String[] args) {
6
7        String inputStr = "abc00123xyz456_0";  // Input String for matching
8        String regexStr = "[0-9]+";            // Regex to be matched
9
10       // Step 1: Compile a regex via static method Pattern.compile(), default is case-sensitive
11       Pattern pattern = Pattern.compile(regexStr);
12       // Pattern.compile(regex, Pattern.CASE_INSENSITIVE);  // for case-insensitive matching
13
14       // Step 2: Allocate a matching engine from the compiled regex pattern,
15       //         and bind to the input string
16       Matcher matcher = pattern.matcher(inputStr);
```

```
17
18        // Step 3: Perform matching and Process the matching results
19        // Try Matcher.find(), which finds the next match
20        while (matcher.find()) {
21           System.out.println("find() found substring \"" + matcher.group()
22                 + "\" starting at index " + matcher.start()
23                 + " and ending at index " + matcher.end());
24        }
25
26        // Try Matcher.matches(), which tries to match the ENTIRE input (^...$)
27        if (matcher.matches()) {
28           System.out.println("matches() found substring \"" + matcher.group()
29                 + "\" starting at index " + matcher.start()
30                 + " and ending at index " + matcher.end());
31        } else {
32           System.out.println("matches() found nothing");
33        }
34
35        // Try Matcher.lookingAt(), which tries to match from the START of the input (^...)
36        if (matcher.lookingAt()) {
37           System.out.println("lookingAt() found substring \"" + matcher.group()
38                 + "\" starting at index " + matcher.start()
39                 + " and ending at index " + matcher.end());
40        } else {
41           System.out.println("lookingAt() found nothing");
42        }
```

```
43
44         // Try Matcher.replaceFirst(), which replaces the first match
45         String replacementStr = "**";
46         String outputStr = matcher.replaceFirst(replacementStr); // first match only
47         System.out.println(outputStr);
48
49         // Try Matcher.replaceAll(), which replaces all matches
50         replacementStr = "++";
51         outputStr = matcher.replaceAll(replacementStr); // all matches
52         System.out.println(outputStr);
53     }
54  }
```

The output is:

```
find() found substring "00123" starting at index 3 and ending at index 8
find() found substring "456" starting at index 11 and ending at index 14
find() found substring "0" starting at index 15 and ending at index 16
matches() found nothing
lookingAt() found nothing
abc**xyz456_0
abc++xyz++_++
```

1.4 Example: Full Numeric Strings `^[0-9]+$` or `^\d+$`

1. The leading `^` and the trailing `$` are known as *position anchors*, which match the start and end positions of the line, respectively. As the result, the entire input string shall be matched fully, instead of a portion of the input string (substring).
2. This regex matches any non-empty numeric strings (comprising of digits 0 to 9), e.g., "`0`" and "`12345`". It does not match with "" (empty string), "`abc`", "`a123`", "`abc123xyz`", etc. However, it also matches "`000`", "`0123`" and "`0001`" with leading zeros.

1.5 Example: Positive Integer Literals `[1-9][0-9]*|0` or `[1-9]\d*|0`

1. `[1-9]` matches any character between 1 to 9; `[0-9]*` matches zero or more digits. The `*` is an *occurrence indicator* representing zero or more occurrences. Together, `[1-9][0-9]*` matches any numbers without a leading zero.
2. `|` represents the OR operator; which is used to include the number `0`.
3. This expression matches "`0`" and "`123`"; but does not match "`000`" and "`0123`" (but see below).
4. You can replace `[0-9]` by metacharacter `\d`, but not `[1-9]`.
5. We did not use *position anchors* `^` and `$` in this regex. Hence, it can match any parts of the input string. For examples,
    a. If the input string is "`abc123xyz`", it matches the substring "`123`".
    b. If the input string is "`abcxyz`", it matches nothing.
    c. If the input string is "`abc123xyz456_0`", it matches substrings "`123`", "`456`" and "`0`" (three matches).
    d. If the input string is "`0012300`", it matches substrings: "`0`", "`0`" and "`12300`" (three matches)!!!

1.6 Example: Full Integer Literals `^[+-]?[1-9][0-9]*|0$` or `^[+-]?[1-9]\d*|0$`

1. This regex match an Integer literal (for entire string with the *position anchors*), both positive, negative and zero.
2. `[+-]` matches either `+` or `-` sign. `?` is an *occurrence indicator* denoting 0 or 1 occurrence, i.e. optional. Hence, `[+-]?` matches an optional leading `+` or `-` sign.
3. We have covered three occurrence indicators: `+` for one or more, `*` for zero or more, and `?` for zero or one.

1.7 Example: Identifiers (or Names) `[a-zA-Z_][0-9a-zA-Z_]*` or `[a-zA-Z_]\w*`

1. Begin with one letters or underscore, followed by zero or more digits, letters and underscore.
2. You can use *metacharacter* `\w` for a word character `[a-zA-Z0-9_]`. Recall that *metacharacter* `\d` can be used for a digit `[0-9]`.

1.8 Example: Image Filenames `^\w+\.(gif|png|jpg|jpeg)$`

1. The *position anchors* `^` and `$` match the beginning and the ending of the input string, respectively. That is, this regex shall match the entire input string, instead of a part of the input string (substring).
2. `\w+` matches one or more word characters (same as `[a-zA-Z0-9_]+`).
3. `\.` matches the dot `(.)` character. We need to use `\.` to represent `.` as `.` has special meaning in regex. The `\` is known as the escape code, which restore the original literal meaning of the following character. Similarly, `*`, `+`, `?` (occurrence indicators), `^`, `$` (position anchors) have special meaning in regex. You need to use an escape code to match with these characters.
4. `(gif|png|jpg|jpeg)` matches either "`gif`", "`png`", "`jpg`" or "`jpeg`". The `|` denotes "OR" operator. The parentheses are used for grouping the selections.
5. The *modifier* `i` after the regex specifies case-insensitive matching (applicable to some languages like Perl and JavaScript only). That is, it accepts "`test.GIF`" and "`TesT.Gif`".

1.9 Example: Email Addresses `^\w+([.-]?\w+)*@\w+([.-]?\w+)*(\.\w{2,3})+$`

1. The *position anchors* `^` and `$` match the beginning and the ending of the input string, respectively. That is, this regex shall match the entire input string, instead of a part of the input string (substring).
2. `\w+` matches 1 or more word characters (same as `[a-zA-Z0-9_]+`).
3. `[.-]?` matches an optional character `.` or `-`. Although dot (`.`) has special meaning in regex, in a character class (square brackets) any characters except `^`, `-`, `]` or `\` is a literal, and do not require escape sequence.
4. `([.-]?\w+)*` matches 0 or more occurrences of `[.-]?\w+`.
5. The sub-expression `\w+([.-]?\w+)*` is used to match the username in the email, before the `@` sign. It begins with at least one word character `[a-zA-Z0-9_]`, followed by more word characters or `.` or `-`. However, a `.` or `-` must follow by a word character `[a-zA-Z0-9_]`. That is, the input string cannot begin with `.` or `-`; and cannot contain `".."`, `"--"`, `".-"` or `"-."`. Example of valid string are `"a.1-2-3"`.
6. The `@` matches itself. In regex, all characters other than those having special meanings matches itself, e.g., `a` matches `a`, `b` matches `b`, and etc.
7. Again, the sub-expression `\w+([.-]?\w+)*` is used to match the email domain name, with the same pattern as the username described above.
8. The sub-expression `\.\w{2,3}` matches a `.` followed by two or three word characters, e.g., `".com"`, `".edu"`, `".us"`, `".uk"`, `".co"`.
9. `(\.\w{2,3})+` specifies that the above sub-expression could occur one or more times, e.g., `".com"`, `".co.uk"`, `".edu.sg"` etc.

Exercise: Interpret this regex, which provide another representation of email address: `^[\w\-\.\+]+\@[a-zA-Z0-9\.\-]+\.[a-zA-z0-9]{2,4}$`.

Code Example in Java

Java keeps the parenthesized back references in `$1`, `$2`, ...

```
1  import java.util.regex.Pattern;
2  import java.util.regex.Matcher;
3
4  public class TestRegexSwapWords {
5     public static void main(String[] args) {
6        String inputStr = "apple orange";
7        String regexStr = "^(\\S+)\\s+(\\S+)$";  // Regex pattern to be matched
8        String replacementStr = "$2 $1";         // Replacement pattern with back references
9
10       // Step 1: Allocate a Pattern object to compile a regex
11       Pattern pattern = Pattern.compile(regexStr);
12
13       // Step 2: Allocate a Matcher object from the Pattern, and provide the input
14       Matcher matcher = pattern.matcher(inputStr);
15
16       // Step 3: Perform the matching and process the matching result
17       String outputStr = matcher.replaceFirst(replacementStr); // first match only
18       System.out.println(outputStr);   // Output: orange apple
19    }
20 }
```

2.8 Occurrence Indicators (Repetition Operators): +, *, ?, {m}, {m,n}, {m,}

A regex sub-expression may be followed by an *occurrence indicator* (aka *repetition operator*):

- `?`: The preceding item is optional and matched at most once (i.e., occurs 0 or 1 times or optional).
- *: The preceding item will be matched zero or more times, i.e., `0+`
- +: The preceding item will be matched one or more times, i.e., `1+`
- `{m}`: The preceding item is matched exactly m times.
- `{m,}`: The preceding item is matched m or more times, i.e., `m+`
- `{m,n}`: The preceding item is matched at least m times, but not more than n times.

For example: The regex xy`{2,4}` accepts "xyy", "xyyy" and "xyyyy".

Lesson 11: Match groups

Regular expressions allow us to not just match text but also to **extract information for further processing**. This is done by defining **groups of characters** and capturing them using the special parentheses **(** and **)** metacharacters. Any subpattern inside a pair of parentheses will be **captured** as a group. In practice, this can be used to extract information like phone numbers or emails from all sorts of data.

Imagine for example that you had a command line tool to list all the image files you have in the cloud. You could then use a pattern such as **^(IMG\d+\.png)$** to capture and extract the full filename, but if you only wanted to capture the filename without the extension, you could use the pattern **^(IMG\d+)\.png$** which only captures the part before the period.

Go ahead and try to use this to write a regular expression that matches only the filenames (not including extension) of the PDF files below.

Exercise 11: Matching Groups

Task	Text	Capture Groups
Capture	file_record_transcript.pdf	file_record_transcript
Capture	file_07241999.pdf	file_07241999
Skip	testfile_fake.pdf.tmp	

Non-capturing groups with ?:

Sometimes we need parentheses to correctly apply a quantifier, but we don't want their contents in the array.

A group may be excluded by adding `?:` in the beginning.

For instance, if we want to find `(go)+`, but don't want to put remember the contents (`go`) in a separate array item, we can write: `(?:go)+`.

In the example below we only get the name "John" as a separate member of the `results` array:

```
let str = "Gogo John!";
// exclude Gogo from capturing
let reg = /(?:go)+ (\w+)/i;

let result = str.match(reg);

alert( result.length ); // 2
alert( result[1] ); // John
```