Regular Expressions

1. Introduction

- A regular expression is a set of special characters that define a pattern.
- They are a type of language that is intended for the matching and manipulation of text.
- In web development they are commonly used to test whether a user's input matches a predictable sequence of characters, such as those in a phone number, postal or zip code, or email address.
- Regular expressions are a concise way to eliminate the conditional logic that would be necessary to ensure that input data follows a specific format.
- Consider a postal code: in Canada a postal code is a letter, followed by a digit, followed by a letter, followed by an optional space or dash, followed by number, letter, and number. Using if statements, this would require many nested conditionals (or a single if with a very complex expression). But using regular expressions, this pattern check can be done using a single concise function call.

1.1 Regular Expression Syntax

- A regular expression consists of two types of characters: **literals** and **metacharacters**.
- A literal is just a character you wish to match in the target (i.e., the text that you are searching within).
- A metacharacter is a special symbol that acts as a command to the regular expression parser.
- There are 14 metacharacters described in the php implementation:

$$. [ ] \ ( ) \wedge \$ | * ? \{ \} +$$

- To use a metacharacter as a literal, you will need to escape it by prefacing it with a backslash (\).
- Table 1 shows Common Regular Expression Patterns.

Pattern	Description
^ qwerty $	If used at the very start and end of the regular expression, it means that the entire string (and not just a substring) must match the rest of the regular expression contained between the ^ and the $ symbols.
\t	Matches a tab character.
\n	Matches a new-line character.
.	Matches any character other than \n.
[qwerty]	Matches any single character of the set contained within the brackets.
[^qwerty]	Matches any single character not contained within the brackets.
[a-z]	Matches any single character within range of characters.
\w	Matches any word character. Equivalent to [a-zA-Z0-9].
\W	Matches any nonword character.
\s	Matches any white-space character.
\S	Matches any nonwhite-space character.
\d	Matches any digit.
\D	Matches any nondigit.
*	Indicates zero or more matches.
+	Indicates one or more matches.
?	Indicates zero or one match.
{n}	Indicates exactly n matches.
{n,}	Indicates n or more matches.
{n, m}	Indicates at least n but no more than m matches.
|	Matches any one of the terms separated by the | character. Equivalent to Boolean OR.
()	Groups a subexpression. Grouping can make a regular expression easier to understand.

- In PHP, regular expressions are contained within forward slashes. So, for instance, to define a regular expression, you would use the following:

```
$pattern = '/ran/';
```

- It should be noted that regular expression pattern checks are case sensitive.
- This regular expression will find matches in all three of the following strings:

```
'randy connolly'
'Sue ran to the store'
'I would like a cranberry'
```

- To perform the pattern check in **PHP**, you would write something similar to the following:

```
$pattern = '/ran/';
$check = 'Sue ran to the store';
if ( preg_match($pattern, $check) ) {
  echo 'Match found!';
}
```

- To perform the same pattern check in **JavaScript**, you would write something similar to the following:

```
var pattern =  /ran/;
if ( pattern.test('Sue ran to the store') ) {
  document.write('Match found!');
}
```

- In JavaScript a regular expression is its own data type. Just as a string literal begins and ends with quote characters, in JavaScript, a regular expression literal begins and ends with forward slashes.

Example#1

The best way to understand regular expressions is to work through the creation of one. For instance, if we wished to define a regular expression that would match a **North American phone number without the area code**, we would need one that matches **any string that contains three numbers, followed by a dash, followed by four numbers without any other character**. The regular expression for this would be:

```
^\d{3}-\d{4}$
```

In this example, the **dash** is a literal character; the rest are all metacharacters. The **^** and **$** symbol indicate the beginning and end of the string, respectively; they indicate that the entire string (and not a substring) can only contain that specified by the rest of the metacharacters.

The metacharacter **\d** indicates a **digit**, while the metacharacters **{3}** and **{4}** indicate **three** and **four** repetitions of the previous match (i.e., a digit), respectively.

A more sophisticated regular expression for a phone number would not allow the first digit in the phone number to be a zero (“0”) or a one (“1”). The modified regular expression for this would be:

```
^[2-9]\d{2}-\d{4}$
```

The [2-9] metacharacter indicates that the first character must be a digit within the range 2 through 9.

We can make our regular expression a bit more flexible by allowing either a single space (440 6061), a period (440.6061), or a dash (440-6061) between the two sets of numbers. We can do this via the [] metacharacter:

```
^[2-9]\d{2}[-\s\.]\d{4}$
```

This expression indicates that the fourth character in the input must match one of the three characters contained within the square brackets (- matches a dash, \s matches a white space, and \. matches a period). We must use the escape character for the dash and period, since they have a metacharacter meaning when used within the square brackets.

If we want to allow multiple spaces (but only a single dash or period) in our phone, we can modify the regular expression as follows.

```
^[2-9]\d{2}[-\s\.]\s*\d{4}$
```