SoftServe | Empowering your Business through Software Development

Module 8:
Regular Expressions

Agenda

- What is "Regular Expression"?
- Creating and running regular expressions in JavaScript
- Constructing regular expressions:
  - Part I: Exact and character set match, basic special characters
  - Part II: Quantifiers, controlling greedy and non-greedy capturing, capturing groups and logical operators
- Useful links

What is "Regular Expression"?

Concept of Regular Expressions

- Regular Expression (Regexp or Regex) is a special sequence of characters that forms a search pattern.
- The concept of Regex has been created in 1950s by American mathematician Stephen Kleene who formalized the description of a regular language.
- Now Regexes are widely used to verify or extract required data and much more

Understanding basics

- Rexes string looks like "**`cent(er|re)`**"
- Each character in Regex may be one of two types:
  1. Regular character with its literal meaning
  2. Special metacharacter with special meaning
- In Regex "**`cent(er|re)`**" characters colored with green are regular characters with literal meaning while colored with red are metacharacters with special meaning
- The Regex in example matches word "center" or "centre" in American or British spelling

Creating and running regular expressions in JavaScript

Create RegExp object

JavaScript has special object RexExp

There are two ways to create an instance of it:

1. `var myRE = new RegExp('SomeExpression');`
2. `var myRE = /SomeExpression/;`

Important note: Variant 2 is preferred because special characters for string formatting are ignored here

RegExp flags

Regular expressions have four optional flags that allow for global and case insensitive searching.

- To indicate a **global** search, use the **g** flag.
- To indicate a **case-insensitive** search, use the **i** flag.
- To indicate a **multi-line** search, use the **m** flag.
- To perform a **"sticky"** search, that matches starting at the current position in the target string, use the **y** flag.

These flags can be used separately or together in any order, and are included as part of the regular expression.

To include a flag with the regular expression, use this syntax:

```
var re = /pattern/flags;
```

or

```
var re = new RegExp("pattern", "flags");
```

Note that the flags are an integral part of a regular expression. They cannot be added or removed later.

RegExp method test()

To check if RegExp matches the string we should use `test()` method of RegExp object. This method accepts string and returns true if it finds a match, otherwise it returns false.

Example

Search a string for the character "o":

```
var str = "Hello";
var re = /o/;
var result = re.test(str);
```

Variable "result" is true

String method search()

To find index in a string which corresponds match for regular expression we should use method search() of a String object.

If successful, search returns the index of the first match of the regular expression inside the string. Otherwise, it returns -1.

Example

Find a position of the character "o" in a string:

```
var str = "Hello";
var re = /o/;
var result = str.search(re);
```

Variable "result" is 4

String method match()

To extract all matches of regular expression from a string, use method match() of a String object. It accepts RegExp as a parameter and returns an array containing all matches or null if there where no matches.

Syntax:

str.match(regexp);

Important note: regular expression should include "g" flag, otherwise method will work same way as RegExp.exec() method which require a loop to get all matches.

Example

Extract all characters "o" from a string:

```
var str = "Hello World!";
var re = /o/g;
var result = str.match(re);
```

Variable "result" is ["o", "o"]

Using Regexes with HTML Forms

HTML5 supports special attribute **pattern** for **<input>** elements of a form.

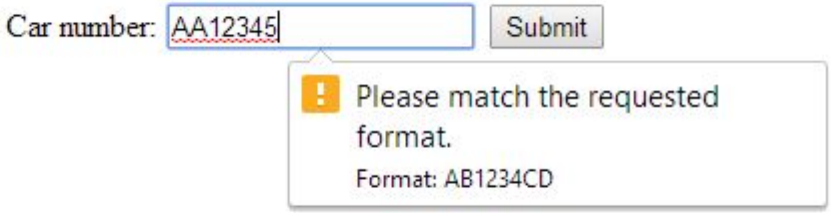
The pattern attribute specifies a regular expression that the <input> element's value is checked against.

The pattern attribute works with the following input types: text, search, url, tel, email, and password.

Use the title attribute to describe the pattern to help the user.

Example form:

```
<form>
  Car number: <input type="text" name="car"
   pattern="[A-Z]{2}\d{4}[A-Z]{2}"
   title="Format: AB1234CD">
   <input type="submit">
</form>
```


Car number: AA12345

Please match the requested format.

Format: AB1234CD

Constructing Regular Expressions

Exact and character set match,
basic special characters

Exact match and anchors

- Checking for EXACT match of some pattern string inside ANY place of the test string:

```
var re = /pattern/;
re.test("some string to test pattern inside"); // true
re.test("there is no test string inside'); // false
```

- Checking for EXACT match of pattern string from the BEGINNING of the test string:

```
var re = /^pattern/;
re.test("pattern starts string"); // true
re.test("there is no pattern at the beginning"); // false
```

- Checking for EXACT match of pattern string at the END of the test string:

```
var re = /pattern$/;
re.test("string ends with pattern"); // true
re.test("there is no pattern at the end"); // false
```

Character set match

- Checking for ONE OR MORE symbols at ANY place in ANY order of the test string:

```
var re = /[abc]/;
re.test("bac"); // true
re.test("baac"); // true
re.test("bdac"); // true
re.test("fpirufieuhfa"); // true
re.test("sdfgsdfg"); // false
```

- SAME but only at the BEGINNING:

```
var re = /^[abc]/;
re.test("bac"); // true
re.test("baac"); // true
re.test("fpirufieuhfa"); // false
re.test("sdfgsdfg"); false
```

- SAME idea at the END

Negated character set

- Using caret symbol "^" as first character set symbol "negates" it, change meaning to opposite:

```
var re = /[^abc]/;
re.test("bac"); // false
re.test("baac"); // false
re.test("bdac"); // true
re.test("fpirufieuhfa"); // true
re.test("sdfgsdfg"); // true
```

Special character: \

- A backslash that precedes a non-special character indicates that the next character is special and is not to be interpreted literally.
  For example, a 'b' without a preceding '\' generally matches lowercase 'b's wherever they occur. But a '\b' by itself doesn't match any character; it forms the special word boundary character.
- A backslash that precedes a special character indicates that the next character is not special and should be interpreted literally. For example, the pattern /a*/ relies on the special character '*' to match 0 or more a's. By contrast, the pattern /a*/ removes the specialness of the '*' to enable matches with strings like 'a*'.
- Do not forget to escape \ itself while using the RegExp("pattern") notation because \ is also an escape character in strings.

Symbol ranges

- How to specify symbol ranges?

```
var re = /[a-z]/; // from a to z
var re = /[a-zA-Z]// from a to Z
var re = /[0-9]/; // from 0 to 9
```

- Using special characters for symbol ranges:
  \w - Matches any alphanumeric character including the underscore.
  \W - Matches any non-word character. Equivalent to [ˆA-Za-z0-9_].

```
/\w/ == /[a-zA-Z0-9_]/
/\W/ == /[^A-Za-z0-9_]/
/\d/ == /[0-9]/
```

Pattern .

- (The decimal point) matches any single character except the newline character.
- For example, /.n/ matches 'an' and 'on' in "nay, an apple is on the tree", but not 'nay'.

Constructing Regular Expressions

Part II: Quantifiers, controlling greedy and non-greedy capturing, capturing groups and logical operators

Quantifiers and greedy capturing

- Quantifiers show how many times preceding symbol should appear in the string:
  - Use curved brackets with one number inside to show how many times exactly symbol should appear: "{4}" – means "4 times exactly"
  - Use curved brackets with pair of numbers like "{0, 3}" to show minimum and maximum number of times (from zero to three)
  - Use special quantifier symbols like "*", "+", "?" (explained later)
- By default, RegExp engine behaves in greedy way and tries to capture as many symbols as possible to match the expression. If we want to capture fewest symbols possible we, we should add question mark after quantifier symbol (explained later).

Quantifier *

- Matches the preceding character 0 or more times. Equivalent to {0,}.
- For example, /bo*/ matches 'boooo' in "A ghost booooed" and 'b' in "A bird warbled", but nothing in "A goat grunted".

```
var re = /bo*/;
re.test("A ghost booooed"); // true
re.test("A bird warbled"); // true
re.test("A goat grunted"); // false
```

Quantifier+

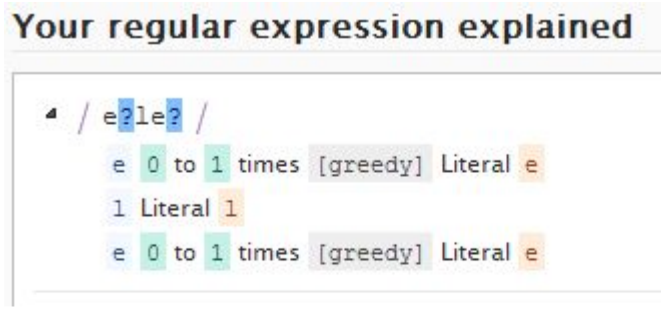
- Matches the preceding character 1 or more times. Equivalent to {1,}.

```
var re = /bo+/; // OR var re = /bo{1,}/
re.test("A ghost booooed"); // true
re.test("A bird warbled"); // false
re.test("A goat grunted"); // false
re.test("A boat sank"); // true
```

Quantifier ? and switching to non-greedy

- Matches the preceding character 0 or 1 time. Equivalent to {0,1}.

```
var re = /e?le?/;
re.test("angel"); // true (el)
re.test("angle"); // true (le)
re.test("oslo"); // true
```


Your regular expression explained

```
/ e?le? /
    e 0 to 1 times [greedy] Literal e
    l Literal l
    e 0 to 1 times [greedy] Literal e
```

- If used immediately after any of the quantifiers *, +, ?, or {}, makes the quantifier non-greedy (matching the fewest possible characters), as opposed to the default, which is greedy (matching as many characters as possible). For example, applying /\d+/ to "123abc" matches "123". But applying /\d+?/ to that same string matches only the "1".

Capturing groups and logical operators

- Round brackets "(",")" used to define capturing groups which mean a sub-expression inside regular expression
- Often used with logical OR operator "|"
- Example:

```
var str = "One man but many men";
var re = /m(a|e)n/g;
var result = str.match(re);
```

Variable "result" is ["man", "man", "men"]

Useful links

- RegEx101: http://regex101.com
- Regular Expressions on Mozilla Developer Network: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions
- RexEgg: http://www.rexegg.com/

Thank you!