Regular Expressions in Javascript

Introduction

1. A **regular expression** is an object in Javascript that describes a pattern of characters. It allows ways of searching or checking for patterns of characters.
2. Regular expressions are often extremely dense and cryptic in appearance. In fact, it is probably best to err on the side of caution in one's use of them, because they are very rigid and are often more difficult to read than to write, making it difficult for subsequent coders to understand. Best to use with good commenting!
3. On the other hand, they can be learned and implemented a little at a time - if one is comfortable with a use, and it makes the solution easier, go for it; if it is too difficult, just ignore them.
4. Regular expressions usually have a significant performance advantage over equivalent string operations in Javascript (according to *Javascript, The Good Parts, page 65*).

   Example: *Create a function SimpleSymbols(str) to take the str parameter being passed and determine if it is an acceptable sequence by either returning 'true' or 'false' (as a string). The str parameter will be composed of + and = symbols with several letters between them (i.e. ++d+===+c++==a) and for the string to be true each letter must be surrounded by a + symbol. So the previous example would return false, as a is not followed by a +. The string will not be empty and will have at least one letter.*

   This problem is a good example of how regular expressions can make life much easier. Without them, the syntax of the solution would be considerably more involved and error prone.

```
function SimpleSymbols(str) {

    //check to see if the string begins or ends with a letter(which would
violate the condition of having a '+' on each side)

    if (/^[a-zA-Z]/.test(str) || /[a-zA-Z]$/.test(str)) {
        return 'false'
    }

    //then check to see if there are any instances of a letter i) preceded
by something other than a '+' or ii) followed by something other than a
'+'

    else if (/[^\+][a-zA-Z]/.test(str) || /[a-zA-Z][^\+]/.test(str)) {
        return 'false'
    }

    else {
        return 'true'
    }
}
```

Where to Use Regular Expressions

RegExp Methods

1. RegExp is an object type in javascript, just as Array, String, Date, etc. RegExp has two methods that are commonly used, exec() and test(). Test() is the easier method, so let's begin with it:

test()

1. Test() is an RegExp method that checks a string to see if it contains the pattern set forth in the given regular expression.
    a. It takes a single parameter, being the string to be tested for a match.
    b. It has a boolean return value, i.e., true if a match occurs, and false if no match occurs.
    c. It has no side effects.

A simple example:

```
var string = "Now is the winter of our discontent made glorious summer"

//below are two regular expressions, the first checks for the combo "ow":
var pattern1 = /ow/
//and the second looks for the pattern "cow":
var pattern2 = /cow/

console.log(pattern1.test(string)) => true
console.log(pattern2.test(string)) => false
```

Test() is a great method to use for practicing regular expressions. As you learn the syntax, make up strings, RegExp objects, and see if they behave as you think they should.

exec()

1. Exec() is a more complicated method.

String Methods

1. There are several string methods that can use regular expressions as arguments. They will also take (or automatically convert) a string as an argument. Each is listed below, with a description and an example. I suggest skipping to the section on constructing regular expressions, working just with the "test()" method until comfortable, then coming back the the following (as well as to "exec()"), and testing them out.

search()

1. Search() is a string method that checks a string to see if it contains a designated item.
    a. it takes a single parameter, being a regular expression. However, if a string is included as the argument rather than a regular expression, it will convert the string to a regular expression.
    b. it returns the index of the first match. If no match is found, it returns -1.
    c. it has no side effects.
    d. compare with indexOf(), which will only take a string as a parameter.

```
 var str = "Now is the winter of our discontent made glorious
summer";
 pattern1 = /Discontent/;
 pattern2 = /discontent/;
 console.log(str.search(pattern1)); => -1
 console.log(str.search(pattern2)); => 25
```

replace()

1. Replace() is a string method that checks a string to see if it contains a specified value, then replaces that value with a designated value. **Note:** It only replaces the first occurence of the found string, unless the search term is a regular expression with a global flag.
    a. it takes two required parameters, a search value consisting of a regular expression or a string and the value with which to replace any located items.
    b. it returns a new string with the substitution(s) made.
    c. it has no side effects.

```
 var str = 'Mr. Blue has a blue guitar and a blue car.'
 pattern1 = 'blue';
 pattern2 = /blue/g;
 newItem = "red";

 console.log(str.replace(pattern1, newItem)); => Mr. Blue has a red
guitar and a blue car.
 console.log(str.replace(pattern2, newItem)); => Mr. Blue has a red
guitar and a red car.
```

split()

1. Split() is a string method that chops up the string according to a given separator value, and returns the parts in an array. **Note:** the separator item ends up getting removed from the array.
    a. it takes two parameters, the first being the separator, which can be a string or regular expression. The second is optional, being a number specifying a maximum number of array items.
    b. it returns an array containing the portions of the split string.
    c. it has no side effects.

```
 var str = 'Mr. Blue has a blue guitar and a blue car.'
 //find occurences of a single letter preceded and followed by a
space.
 var pattern1 = /\s\w\s/;
 var arr = str.split(pattern1);
 console.log(arr) => ['Mr. Blue has', 'blue guitar and', 'blue
car.']
```

match()

1. Match is a string method that searches the object string for matches to a regular expression.
    a. it takes a single parameter, being a regular expression. It is important to make certain that the global flag (g) is used if one wants all the matches.
    b. it returns an array. If the global flag is included in the regular expression, it returns an array of all the matching items. If the global flag is not included, it returns an array containing the first match, the index of the match in the object string, and the entire object string.
    c. it has no side effects.

```
 var str = "The rain in Spain falls mainly on the captain.";
 var pattern1 = /ain/;
 //add the global flag
 var pattern2 = /ain/g;
 var arr1 = str.match(pattern1);
 var arr2 = str.match(pattern2);

 console.log(arr1) => ['ain', index: 5, input: 'The rain in Spain
falls mainly on the captain.']

 console.log(arr2) => ['ain', 'ain', 'ain', 'ain']
```

How to Make Them

1. There are two ways to make a regular expression object, by using a RegExp constructor, or by literal notation. Using the constructor can be useful in some circumstances, particularly where one wishes to create a regular expression dynamically. However, it is far more common to see them created by literal notation, and after a brief discussion of the constructor, this outline will use literal notation exclusively.

The RegExp Constructor

1. One can create a new regular expression in javascript using the **RegExp constructor**. To do so, follow this pattern:

```
var pattern = new RegExp('[expression]', '[flags]')
```

2. Don't forget the final 'p' - it is 'RegExp', not 'RegEx'!
3. The following is an example of the use of the constructor, in which a user is asked what word to search for in a string:

```
    var str = 'The rain in Spain falls mainly in the rainy zone!";
    var input = 'rain';
    //create the regular expression based on the input
    pattern = new RegExp(input, 'g');
    var arr = str.match(pattern)
    console.log(arr) => ['rain', 'rain]
```

Literal Notation

1. Regular expressions are denoted by the **forward slashes**. In addition, the regular expression may be immediately followed by any of three flags ('i', 'g', or 'm'), which will be covered shortly. The following is a declaration of a simple regular expression that matches the word sequence of letters 'to':

```
var regex = /to/
```

2. If we wanted to find 'To', 'TO', or 'tO' as well as 'to', we would follow it with the 'i' flag to make it case insensitive, as follows:

```
 var regex = /to/i
```

Regular Expression Syntax

1. The following section goes through a number of the symbols used to form regular expressions. In order to practice them, I will (for the most part), provide examples using

the test() method described above, as it is very simple to use and delivers a easy to understand true/false response. In addition, the following makes use of literal notation exclusively, but any of the following could be accomplished using the RegExp constructor.

2. One can delay the inevitable by talking *about* regular expressions, complaining about them, or pretending to understand them in theory. However, to be able to use them (whether in writing code or, even more challenging, reviewing previously written code), one must at some point simply memorize a rather modest amount of syntax. I suggest two approaches for doing this: first, make flash cards, a dual column list, or some other means of concretely testing one's memory, then go through the cards once a day until they are solidly in memory. Second, use them in code, even if only in small snippets. The only way to really learn how they work is to actually use them.

Rule 1: Begin and End the Regular Expression With "/"

Rule 2: A regular expression consisting of only of simple char values merely matches the target against that exact string.

1. Although not used when using the RegExp constructor, the more typical way one will see regular expressions is as a string of characters beginning and ending with a forward slash.
2. The easiest example of a regular expression is one in which there is nothing going on except a simple search. Of course, this does not add any functionality to simply using indexOf() with a string, but it is a place to start: Examples:

```
var pattern = /dog/;
var strings = ["cat", "dog", "doggie", "dig", "Thou shalt have no other
Dogs before me"];
var results = strings.map(function(val){
    return pattern.test(val);
});

console.log(results) // => [false, true, true, false, false]


var pattern = /3.14/;
var string = "The value of many numbers cannot be expressed in a finite
number of decimal digits.  For example, the value of pi is only
approximated by the digits 3.1415 and even rational fractions often result
in unending decimal representations."

console.log(pattern.test(string)) // => true
```

Rule 3: A "." is a wild card character that represents any other character (except line terminators).

1. In many search contexts, "*" is the wild card character. It has another purpose in regular exprsessions, so keep them separate. Examples:

```
var pattern = /h.t/;
var strings = ['hat', 'http', 'hot', 'the', 'hetero', 'hint', 'h#t'];
var res = strings.map(function(val){
    return pattern.test(val);
})

console.log(res); // => [true, true, true, false, true, false, true]
```

Rule 4: A "\w" denotes a word-class character. A "\W" denotes anything but.

1. A "word class" character is a letter, a numeric digit, and an underscore. A backslash and lower case "w" is basically a wild card whose wildness is limited to these characters.
2. A backslash followed by an upper case "W" indicates a non-word-class character, basically a wildcard whose wildness is limited to anything *other than* letters, numeric digits, and underscores.

3. Note the following examples:

```
var pattern = /\w\w\w\W\w\w\w\W\w\w\w\w/
var strings = ['757-425-6200', '757_425_6300', 'Why she left'];

var res = strings.map(function(val){
    return pattern.test(val);
});

console.log(res); => [true, false, true];
```

Rule 5: A "\d" denotes the digits class. A "\D" denotes anything but.

1. Similar to the use of the "\w" above, a backslash followed by a lower-case "d" indicates a digit character (0123456789), and an upper-case "D" denotes any character *other than* a numeric digit.
2. Note that these are **not numbers**, but string characters that are digits.

```
var pattern = /\d\d\d\D\d\d\d\D\d\d\d\d/
var strings = ['757-425-6200', '757_425_6300', 'Why she left'];

var res = strings.map(function(val){
    return pattern.test(val);
});

console.log(res); => [true, true, false]
```

Rule 6: A "\s" denotes a whitespace class character. A "\S" denotes anything but.

1. Similar to "\w", and "\d", a backslash followed by a lower-case "s" indicates a whitespace-class character. These include spaces, tabs, newlines, returns, and several other characters. For a complete listing, see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp
2. Also as above, a backslash followed by an upper-case "S" indicates any character *other than* a whitespace-class character.

```
var pattern = /\S\S\S\s\S\S\S\s\S\S\S\S/
var strings = ['757-425-6200', '757 425 6300', 'Why\nshe\tleft'];

var res = strings.map(function(val){
    return pattern.test(val);
});

console.log(res); // => [false, true, true]
```

Rule 7: Brackets[] are used to provide groups of characters.

1. The groupings above (i.e., \w \d \s \W \D \S) can be useful but are quite broad. We can define other groupings using brackets[].
2. Alternatives can be included inside brackets, without any separating commas or other notation. For example, if we want to search for a vowel, we could search for [aeiou].
3. Dashes can be used inside brackets to indicate a range of characters. For example, if we want to find a lower-case letter, rather than type all twenty-six inside the bracket, we can type [a-z]. To find a digit, we can type [0-9]. To find any letter(but not the underscores or digits that "\w" would pick up), we can type [a-zA-Z].
4. **IMPORTANT:**The symbol ^ means "not" when in is included inside brackets. It has another meaning outside brackets. Do not confuse them. Thus, [^abc] will search for anything *other than* the letters a, b, or c.
5. Also, note that we are still searching for one character at a time. The notation [a-z] searches for a single character that is a lowercase letter.

```
var pattern = /[1-3]/
var strings = ['757-445-6600', '757 425 6300', 'Why\nshe\tleft'];
var res = strings.map(function(val){
    return pattern.test(val);
});
console.log(res); // => [false, true, false]


var pattern = /[BN][uo][sw]/
var strings = ['Now is the winter of our discontent', 'Busing is
controversial', 'Go to Boston'];
var res = strings.map(function(val){
    return pattern.test(val);
});
console.log(res); => [true, true, true]


var pattern = /[^Bn][uo][sw]/
var strings = ['Now is the winter of our discontent', 'Busing is
controversial', 'Lost in Boston'];
var res = strings.map(function(val){
    return pattern.test(val);
});
console.log(res); => [true, false, true]
```

Rules 8: An asterisk (*) causes a match of zero or more of a specified character.

1. As noted before, the * is not strictly a wildcard (which is "."), but acts in a somewhat similar fashion, causing a search for strings having zero or more of the specified item.
2. Note the following examples:

```
var patterns = [/\d[a-z]\d/, /\d[a-z]*\d/, /\d*[a-z]*\d/];
var string = '32test5';
var res = patterns.map(function(val){
    return val.test(string);
});
console.log(res); // => [false, true, true];
```

3. In the above examples, what were the found matches, that allowed the second and third examples to return true?
    a. in the first, there was no match because we needed one digit, followed by one

lower-case letter, followed by one digit. That was not present.
   b. in the second, we had a match, because we had one digit, followed by zero or more lower-case letters, followed by one digit. The match was "2test5".
   c. in the third, we had a match, because we had zero or more digits, followed by zero or more lower-case letters, followed by one digit. The match was "32test5".

Rule 9: A plus (+) causes a match of one or more of a specified character.

1. Technically, the "+" operates very similarly to the "*". However, "+" requires at least one instance of the value to which it is appended. Compare the following examples:

```
var patterns = [/\d+[a-z]\d/, /\d[a-z]+\d/, /\d+[a-z]+\d/];
var string = '32ab5';
var res = patterns.map(function(val){
    return val.test(string);
});
console.log(res); // => [false, true, true];
```

2. In the above examples, the found matches were:
   a. in the first, there was no match, because we needed to have one or more digits, followed by a single lower-case letter, followed by a single digit.
   b. in the second, we did have a match, which was "2ab5".
   c. in the third, the match is a little broader, even though both the second and third return true to the test() method. It is "32ab5".

Rule 10: A question mark (?) causes a match of 0 or 1 of a specified character.

1. A question mark, following a search value, causes a match on zero or one of the specified characters. Thus:

```
var pattern = /d?/
var strings = ['test', 'test1', 'test23423'];
var res = strings.map(function(val){
    return pattern.test(val)
});
console.log(res); // => [true, true, true]
```

2. Note the last result. There are many more than zero or one numbers. However, it tests true, because the match *requires* only zero or one items, it is not restricted to only zero or one. The actual match in the last item is "2," the "3423" are not included in the match,

but their presence doesn't affect the validity of the match.

Rule 11: Brackets following an expression are used to indicate a number of repetitions.

1. The * and + quantifiers are quite broad, covering "0 or more", or "one or more", respectively. For finer control, we can use brackets {} containing numbers. If we wanted to find repetitions of the letter "n", we could do the following:
    a. n{5} would look for five ns in a row.
    b. n{3,5} would look for three, four, or five ns.
    c. n{3,} would look for three or more ns. Thus, n+ is the same as n{1,}.
2. The following example is a much easier to read validation of a telephone number:

```
var pattern = /\d{3}-\d{3}-\d{4}/;
var string = '212-867-5309';
var res = pattern.test(string);
console.log(res); // => true;
```

Rule 12: Unless otherwise noted, search results are "Greedy"". Use "?" to make them "Nongreedy".

1. Because we have been using the test() method to try out the above rules, we have seen only whether or not a search is satisfied. We have not had returned what it is satisfied with (although the last couple of examples have this information below).
2. The exec() method returns the actual match, if the search is successful. Thus, one can see if the search result of /\d+/ in "123" is "1" or "123". Unless otherwise instructed, the search result will be ""greedy", i.e., it will contain as many characters as it can. Thus:

```
var pattern = /\d+/;
var string = "123";
console.log(pattern.exec(string)) // => "123"
```

    This is because the search is for one or more digits, and the biggest result will be all three digits.

3. The question symbol (?), when following a quantifier, converts it from being "greedy" to being "nongreedy", i.e., it is satisfied with the smallest match that will satisfy the

search.**Important:**Don't forget that the question mark has another meaning, that of being a zero or one quantifier. Also, when searching for a question mark, one should use an escape \ to denote it. See the following examples:

```
var pattern = /\d+?/;
var string = "123";
console.log(pattern.exec(string)) // => "1"
```

Rule 13: The search term can be sought at the beginning or end of a string by use of ^ and $, respectively.

1. The symbols ^ and $ are used to mark the beginning and end, respectively, of the search string. For example, if it is important that the very first character of a string is "A", we could begin our search /^A . . ./.
2. **NOTE:**Do not confuse the use of ^ in this location with the use of ^ inside brackets [], where it means "not," as discussed above.
3. By using both the ^ at the beginning and the $ at the end of our regular expression, we can control the entire thing much more closely. For example, if we were validating a telephone number input, we might try (as presented above):

```
var pattern = /\d{3}-\d{3}-\d{4}/;
var string = '212-867-5309';
var res = pattern.test(string);
console.log(res); // => true;
```

4. However, this would also return true if the number entered were '212-867-53094hotsexxx'. But if we include the ^ and $, we can assure that we return true only on values that match, without extraneous material, as illustrated below:

```
var patterns = [/\d{3}-\d{3}-\d{4}/, /^\d{3}-\d{3}-\d{4}$/];
var string = '212-867-53094hotsexxx';
var res = patterns.map(function(val) {
    return val.test(string);
});
console.log(res); // => [true, false];
```

Rule 14: The search term can be sought at the beginning or end of a word by use of

\b, or not at the beginnig or end of a work by use of \B.

1. **Note** that the pattern "[\b]" is used to signify a backspace. Do not mix up with this "\b".
2. The \b can be used in front of, or at the end of, a search term, signifying that it should be found at the beginning or end of a word (i.e., a string bounded by spaces). See the following examples:

```
var string = "Now is the winter of our discontent made glorious summer";
var patterns = [/\bdis/, /nt\b/, /\bnt/, /\b[aeiou]/];
var res = patterns.map(function(val){
    return val.test(string);
})
console.log(res);// => [true, true, false, true];
```

Rule 15: Alternative (logical or) searches can be made using the pipe character (|).

1. The pipe character can be used to provide alternative items to search. For example, if we were looking for a title, but did not know what it would be, the following would check for any of the listed possibilities:

```
var pattern = /Mr\.|Mrs\.|Ms\./|Dr\.;
var strings = ['Telephone message for Mr. Man.', 'Telephone message for
Ms. Kim.', 'Telephone message for Miss Molly.', 'Paging Dr. Watson']

var res = strings.map(function(val){
    return pattern.test(val);
})
console.log(res); => [true, true, false, true];
```

Rule 16: Matches can be made dependent on the presence (?=) or absence(?!) of subsequent items that are not included in the match.

1. This is known as either a positive or negative lookahead. For example:

```
var pattern1 = /aq(?=u)/;
var pattern2 = /aq(?!u)/;
var string = 'aquamarine in Iraq';
var string2 = 'aq uamarine in Iraq';
console.log(pattern1.exec(string)); => ['aq', index: 0];
console.log(pattern2.exec(string)); => ['aq', index: 16];
console.log(pattern2.exec(string2)); => null;
```

2. In the first pattern, we are using a positive lookahead, i.e., looking for instances of "aq," but only where it is immediately followed by the letter 'u'. In the last example, because of the space inserted into the word 'aquamarine' in string2, there is no match. In the second pattern, we are using a negative lookahead, i.e., looking for instances of 'aq' not immediately followed by the letter 'u'.

Flags

1. The main body of the regular expression can be followed by one or more of three flag characters, as follows:
    a. **i** makes the search case-insensitive. Note that it makes the entire search case insensitive. For example, if one is searching for "newtonian" but isn't certain if it will have an upper-case first letter, but wished to exclude any word with upper-case letters other than the first, one should use:

    ```
    var pattern = /[nN]ewtonian/, and not
    var pattern = /newtonian/i
    ```

    b. **g** is for global search. Without a g flag, the search runs until it has a hit. This doesn't make any difference in the test() method, in which the return value switches from false to true on the first occurence and that is it. But in other contexts, it makes a big difference. See the following examples:

```
 var pattern1 = /blue/;
 var pattern2 = /blue/g;
 var string = "My favorite color is blue.  When I become rich, I'm
going to buy a blue car, a blue house, a blue dog, and get a blue
tatoo."
 var res = string.replace(pattern1, 'red');
 var res2 = string.replace(pattern2, 'red');
 console.log(res) // => "My favorite color is red.  When I become
rich, I'm going to buy a blue car, a blue house, a blue dog, and get
a blue tatoo."
 console.log(res2) // => "My favorite color is red.  When I become
rich, I'm going to buy a red car, a red house, a red dog, and get a
red tatoo."
```

c. **m** is for multiline mode. It treats the beginning and end characters (the ^ and $ characters described above, as working for each line, not over the entire input string. Compare the following:

```
 var pattern1 = /^\d{3}-\d{3}-\d{4}$/;
 var pattern2 = /^\d{3}-\d{3}-\d{4}$/m;
 var string = "757-225-1234\n757-345-8765\n123-876-5678"

 var res1 = pattern1.test(string);
 var res2 = pattern2.test(string);

 console.log(res1) // => false
 console.log(res2) // => true;
```