

Day 7: Regular Expressions I

 by AvmnuSng

Regular Expressions in JavaScript

A *Regular Expression*, or *RegExp*, is a *pattern* used to match character combinations in a string. In JavaScript, regular expressions are also objects. We'll start by giving some basic examples, and then explain the syntax needed to construct and understand RegExes in further detail.

Creating a Regular Expression

A regular expression consists of a *pattern string* and, potentially, a *flag* specifying further detail on how the pattern should be matched. We construct regular expressions by using *regular expression literals* or *RegExp* class objects.

Regular Expression Patterns

We generally construct RegExp patterns using the basic characters we wish to match (e.g., `abc`), or a combination of basic and special characters (e.g., `ab*c` or `(\d+)\.\d*`).

Regular Expression Literal

A regular expression literal is a RegExp pattern enclosed within forward slashes:

```
const re = /ab+c/;
```

This RegExp above matches the character `a`, followed by one or more instances of the character `b`, followed by the character `c`.

RegExp Objects

We can write a regular expression string and pass it as an argument to the *RegExp* constructor:

```
const re = new RegExp('ab+c');
```

Flags

To create a *RegExp* object, we use this syntax:

```
new RegExp(pattern[, flags])
```

To create a regular expression literal, we use this syntax:

```
/pattern/flags
```

If specified, flags can have any combination of the following values:

- *g*: global match.

Table Of Contents

[Creating a Regular Expression](#)[Regular Expression Patterns](#)[Flags](#)[Special Characters in Regular Expressions](#)[Character Classes](#)[Character Sets](#)[Match](#)[Search](#)[Split](#)[Example](#)

- *i*: ignore case.
- *m*: multiline. Treats beginning (^) and end (\$) characters as working over multiple lines.
- *u*: unicode. Treat pattern as a sequence of unicode code points.
- *y*: sticky. Matches only from the index indicated by the lastIndex property of this regular expression in the target string.

Special Characters in Regular Expressions

- *Character Classes*
- *Character Sets*
- *Alteration*
- *Boundaries*
- *Grouping and back references*
- *Quantifiers*
- *Assertions*

Character Classes

This is not a class in the traditional sense, but rather a term that refers to a set of one or more characters that can be used to match a single character from some input string. Here are the basic forms:

- Enclosed within square brackets. Specify the what you'd like your expression to match within square brackets; for example, `[a-f]` will match any lowercase `a` through `f` character.
- Predefined: These consist of a backslash character (`\`) followed by a letter. The table below shows some predefined character classes and the characters they match.

Character	Matches
<code>.</code>	The period matches any single character, except line terminators (e.g., a newline).
<code>\d</code>	A single digit character (i.e., <code>[0-9]</code>).
<code>\D</code>	A single non-digit character (i.e., <code>[^0-9]</code>).
<code>\w</code>	A single alphanumeric word character, including the underscore (i.e., <code>[A-Za-z0-9_]</code>).
<code>\W</code>	A single non-word character (i.e., <code>[^A-Za-z0-9_]</code>).
<code>\s</code>	A single whitespace character. This includes space (<code> </code>), tab (<code>\t</code>), form feed, line feed, and other Unicode spaces.
<code>\S</code>	A single non-whitespace character (i.e., <code>[^\w]</code>).

Character Sets

- The character set `[abcd]` will match any one character from the set `{a, b, c, d}`. This is equivalent to `[a-d]`.
- The character set `[^abcd]`: Matches anything other than the enclosed characters. This is equivalent to `[^a-d]`.

Alteration

We use the `|` symbol to match one thing or the other. For example, `a|b` matches either `a` or `b`.

Boundaries

Character	Matches
<code>^</code>	Matches beginning of input. If the multiline flag is set to true, also matches immediately after a line break character.
<code>\$</code>	Matches end of input. If the multiline flag is set to true, also matches immediately before a line break character.
<code>\b</code>	A zero-width word boundary, such as between a letter and a space.
<code>\B</code>	Matches a zero-width non-word boundary, such as between two letters or between two spaces.

Grouping and back references

`(a)`: Matches `a` and remembers the match. These are called capturing groups.

`(?:a)`: Matches `a` but does not remember the match. These are called non-capturing groups.

`\n`: Here `n` is a positive integer. A back reference to the last substring matching the `n` parenthetical in the regular expression.

Quantifiers

`a*`: Matches the preceding item `a`, 0 or more times.

`a+`: Matches the preceding item `a`, 1 or more times.

`a?`: Matches the preceding item `a`, 0 or 1 time.

`a{n}`: Here `n` is a positive integer. Matches exactly `n` occurrences of the preceding item `a`.

`a{n, }`: Here `n` is a positive integer. Matches at least `n` occurrences of the preceding item `a`.

`a{n, m}`: Here `n` and `m` are positive integers. Matches at least `n` and at most `m` occurrences of the preceding item `a`.

Assertions

`a(?=b)`: Matches `a` only if `a` is followed by `b`.

`a(?!b)`: Matches `a` only if `a` is not followed by `b`.

Working with Regular Expressions

Regular expressions are used with the RegExp methods:

- `test`
- `exec`

and with the String methods:

- `match`
- `search`
- `split`
- `replace`

The `test` Method

The `test()` method executes a search for a match between a regular expression and a specified string. Returns true or false.

-	EXAMPLE
1	<code>// Test whether 'learn' is contained at the very beginning of a stri</code>
2	
3	<code>var re = /^learn/;</code>
4	<code>var str1 = 'learn regular expressions';</code>
5	<code>var str2 = 'write regular expressions';</code>
6	

```
7 console.log(re.test(str1));
8 console.log(re.test(str2));
```

Output

```
true
false
```

Run

The exec Method

The `exec()` method executes a search for a match in a specified string. Returns a result array or null.

EXAMPLE

```
1 // Match 'quick brown' followed by 'jumps', ignoring characters in b
2 // Remember 'brown' and 'jumps'
3 // Ignore case
4
5 var re = /quick\s(brown).+?(jumps)/ig;
6 var str = 'The Quick Brown Fox Jumps Over The Lazy Dog.';
7 var res = re.exec(str);
8
9 console.log(res);
10 console.log();
11
12 // The result object contains following information:
13 // 1. [0] is the full string of characters matched
14 // 2. [1], ...[n] is the parenthesized substring matches, if any. Th
15 // 3. index is the 0-based index of the match in the string.
16 // 4. input is the original string.
17
18 console.log('string of characters matched = ' + res[0]);
19 console.log('first parenthesized substring match = ' + res[1]);
20 console.log('second parenthesized substring match = ' + res[2]);
21 console.log('index of the match = ' + res.index);
22 console.log('original string = ' + res.input);
```

Output

```
[ 'Quick Brown Fox Jumps',
  'Brown',
  'Jumps',
  index: 4,
  input: 'The Quick Brown Fox Jumps Over The Lazy Dog.' ]
```

```
string of characters matched = Quick Brown Fox Jumps
first parenthesized substring match = Brown
second parenthesized substring match = Jumps
index of the match = 4
original string = The Quick Brown Fox Jumps Over The Lazy
Dog.
```

Run

Match

The `match()` method retrieves the matches when matching a string against a regular expression.

EXAMPLE

Find 'Chapter', followed by ``1`` or more numeric characters, followed by a decimal point, followed by a zero or more numeric characters, and use a flag to specify that the results are `*case-insensitive*`.

```
1 var re = /see (chapter \d+(\.\d)*)/i;  
2 var str = 'For more information on regular expressions, see Chap  
3  
4 console.log(str.match(re));
```

Output

```
'Chapter 3.4.5.1',  
'1',  
index: 45,  
input: 'For more information on regular expressions, see Cha  
pter 3.4.5.1 and CHAPTER 2.3' ]
```

Run

Search

The `search()` method executes a search for a match between a regular expression and this String object. If successful, `search()` returns the index of the first match of the regular expression inside the string. Otherwise, it returns -1.

EXAMPLE

Check whether or not a string contains the word 'learn'.

```
1 const re = /learn/;  
2 const str1 = 'Today, we\'ll learn about regular expressions.';  
3 const str2 = 'Tomorrow, we\'ll write regular expressions '  
4             + 'and learn some complex expressions.';  
5 const str3 = 'We\'re all done now!';  
6  
7 console.log(str1);  
8 console.log('A search for', re, 'returns', str1.search(re), '\n');  
9 console.log(str2);  
10 console.log('A search for', re, 'returns', str2.search(re), '\n');  
11 console.log(str3);  
12 console.log('A search for', re, 'returns', str3.search(re));
```

Output

```
Today, we'll learn about regular expressions.  
A search for /learn/ returns 13
```

```
Tomorrow, we'll write regular expressions and learn some  
complex expressions.  
A search for /learn/ returns 46
```

```
We're all done now!  
A search for /learn/ returns -1
```

Run

Split

The `split()` method splits a String object into an array of strings by separating the string into substrings. Separator specifies the character(s) to use for separating the string. The separator is treated as a string or a regular expression. If separator is omitted, the array returned contains one element consisting of the entire string. If separator is an empty string, `str` is converted to an array of characters.

-

EXAMPLE

Split a name string at the space separating the first and last names.

```
1 const name = 'Julia Roberts';
2 const res = name.split(' ');
3
4 console.log('The split array:', res);
5 console.log('First Name:', res[0]);
6 console.log('Last Name:', res[1]);
```

Output

Run

Replace

The `replace(pattern, replacement)` method returns a new string where some (or all) occurrences of a matched ***pattern*** have been replaced with a ***replacement*** substring.

- ***pattern***: This value can be a string or a *RegExp* object to match against the calling string.
- ***replacement***: This value can be a substring to replace the match with, or it can be a function to invoke that generates the replacement substring.

-

EXAMPLE

In this example, we replace any occurrence of the substring 'RegExp' with 'Regular Expression'

```
1 const re = /RegExp/;
2 const myString = 'We\'re learning about RegExps.';
3 const replacementString = 'Regular Expression';
4
5 console.log(myString);
6 console.log(myString.replace(re, replacementString));
```

Output

Run

Example

-

EXAMPLE

Find a *substring* of length greater than **1** that starts *and* ends with same character.

```
1 const re = /(.)\1/;
2
3 const str1 = 'wxyz';
4 const str2 = 'wxyzw';
5 const str3 = 'wxyzx';
```

```
6 const str4 = 'wxyz';
7
8 console.log('substring:', str1.match(re));
9 console.log('substring:', str2.match(re)[0]);
10 console.log('substring:', str3.match(re)[0]);
11 console.log('substring:', str4.match(re)[0]);
```

Output

```
substring: null
substring: wxyzw
substring: xyzx
substring: wxyw
```

[Run](#)

Let's break down the regular expression `(.)*\1`:

1. `(.)` captures *any character*.
2. `.*` stipulates that the captured character must followed by zero or more occurrences of any character.
3. `\1` is a backreference to the first *capture group* in our expression (i.e., `(.)`). It stipulates that the character following whatever we matched in step **2** must match whatever was *captured* inside the parentheses in step **1**.

Related challenge for **Regular Expressions in JavaScript**

Day 7: Regular Expressions II



Success Rate: **96.89%** Max Score: **15** Difficulty:

[Solve Challenge](#)