Day 7: Regular Expressions I

by AvmnuSng

Regular Expressions in JavaScript

A *Regular Expression*, or *RegEx*, 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 RegEx 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 RegEx pattern encosed within forward slashes:

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

This RegEx 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');
```

Go to Top

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.
- *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
.	The period matches any single character, except line terminators (e.g., a newline).
\d	A single digit character (i.e., `[0-9]`).
\D	A single non-digit character (i.e., `[^0-9]`).

Go to Top

Character	Matches
`\w`	A single alphanumeric word character, including the underscore (i.e., `[A-Za-z0-9_]`).
`\W`	A single non-word character (i.e., `[^A-Za-z0-9_]`).
`\s`	A single whitespace character. This includes space ( ), tab (`\t`), form feed, line feed, and other Unicode spaces.
`\S`	A single non-whitespace character (i.e., `[^\w]`).

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
`^`	Matches beginning of input. If the multiline flag is set to true, also matches immediately after a line break character.
`$`	Matches end of input. If the multiline flag is set to true, also matches immediately before a line break character.
`\b`	A zero-width word boundary, such as between a letter and a space.
`\B`	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.

Go to Top

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

```
// Test whether 'learn' is contained at the very beginning of a stri

var re = /^learn/;
var str1 = 'learn regular expressions';
var str2 = 'write regular expressions';

console.log(re.test(str1));
```

Output

Run

The `exec` Method

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

EXAMPLE

```
// Match 'quick brown' followed by 'jumps', ignoring characters in b
// Remember 'brown' and 'jumps'
// Ignore case

var re = /quick\s(brown).+?(jumps)/ig;
var str = 'The Quick Brown Fox Jumps Over The Lazy Dog.';
var res = re.exec(str);

console.log(res);
console.log();

// The result object contains following information:
// 1. [0] is the full string of characters matched
// 2. [1], ...[n] is the parenthesized substring matches, if any. Th
// 3. index is the 0-based index of the match in the string.
// 4. input is the original string.

console.log('string of characters matched = ' + res[0]);
console.log('first parenthesized substring match = ' + res[1]);
```

Go to Top

```
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

Run

Match

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

- EXAMPLE

Go to Top

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*.

```
var re = /see (chapter \d+(\.\d)*)/i;
var str = 'For more information on regular expressions, see Chap

console.log(str.match(re));
```

Run

Output

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'.

```
const re = /learn/;
const str1 = 'Today, we\'ll learn about regular expressions.';
const str2 = 'Tomorrow, we\'ll write regular expressions '
             + 'and learn some complex expressions.';
const str3 = 'We\'re all done now!';

console.log(str1);
console.log('A search for', re, 'returns', str1.search(re), '\n');
console.log(str2);
console.log('A search for', re, 'returns', str2.search(re), '\n');
console.log(str3);
```

Output

Run

Split

Go to Top

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]);
```

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);
```

Output

Run

Example

- EXAMPLE

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

```
const re = /(.).*\1/;

const str1 = 'wxyz';
```

Go to Top

```
const str2 = 'wxyzw';
const str3 = 'wxyzx';
const str4 = 'wxywz';

console.log('substring:', str1.match(re));
console.log('substring:', str2.match(re)[0]);
console.log('substring:', str3.match(re)[0]);
console.log('substring:', str4.match(re)[0]);
```

Output

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: 97.20% **Max Score:** 15 **Difficulty:**

Solve Challenge