Regular Expressions

RegExp Object

A regular expression is an object that describes a pattern of characters.

Regular expressions are used to perform pattern-matching and "search-and-replace" functions on text.

Syntax

/pattern/modifiers;

Modifiers Modifiers are used to perform case-insensitive and global searches:

Modifier	**Description**
g	Perform a global match (find all matches rather than stopping after the first match)
i	Perform case-insensitive matching
m	Perform multiline matching

Using a **regular expression literal**, which consists of a pattern enclosed between slashes, as follows:

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

Regular expression literals provide compilation of the regular expression when the script is loaded.

calling the **constructor function** of the RegExp object, as follows:

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

Using the constructor function provides runtime compilation of the regular expression.

```
<html>
<body>
<h2>JavaScript Regular Expressions</h2>
<p>Click the button to do a case-insensitive search for ‘Punjab’
in a string.</p>
<button onclick="myFunction()">Click here</button>
<p id="demo"></p>
<script>
function myFunction() {
  var str = “Welcome to Punjab";
  var patt = /Punjab/i;
  var result = str.match(patt);
  document.getElementById("demo").innerHTML = result;
}
</script>
</body>
</html>
```

- **/punjab/i is a regular expression.**
- **punjab is a pattern (to be used in a search).**
- **i is a modifier (modifies the search to be case-insensitive).**

Brackets

Brackets are used to find a range of characters:

Expression	Description
[abc]	Find any character between the brackets
[^abc]	Find any character NOT between the brackets
[0-9]	Find any character between the brackets (any digit)
[^0-9]	Find any character NOT between the brackets (any non-digit)
(x|y)	Find any of the alternatives specified

JavaScript RegExp [abc] Expression

Do a global search for the character "h" in a string:

```
<html>
<body>

<p>Click the button to do a global search for the character "h" in a string.</p>

<button onclick="myFunction()">Try it</button>

<p id="demo"></p>

<script>
function myFunction() {
  var str = "Is this all there is?";
  var patt1 = /[h]/g;
  var result = str.match(patt1);
  document.getElementById("demo").innerHTML = result;
}
</script>

</body>
</html>
```

Definition and Usage

The [abc] expression is used to find any character between the brackets.

The characters inside the brackets can be any characters or span of characters:

[abcde..] - Any character between the brackets
[A-Z] - Any character from uppercase A to uppercase Z
[a-z] - Any character from lowercase a to lowercase z
[A-z]- Any character from uppercase A to lowercase z

Tip: Use the [^abc] expression to find any character NOT between the brackets.

```
<html>
<body>

<p>Click the button to do a global search for the
character-span [a-h] in a string.</p>

<button onclick="myFunction()">Try it</button>

<p id="demo"></p>

<script>
function myFunction() {
  var str = "Is this all there is?";
  var patt1 = /[a-j]/g;
  var result = str.match(patt1);
  document.getElementById("demo").innerHTML =
result;
}
</script>

</body>
</html>
```

```
<html>
<body>

<p>Click the button to do a global search for any
character from uppercase "A" to lowercase "e".</p>

<button onclick="myFunction()">Try it</button>

<p id="demo"></p>

<script>
function myFunction() {
  var str = "I Scream For Ice Cream, is that OK?!";
  var patt1 = /[A-e]/g;
  var result = str.match(patt1);

document.getElementById("demo").innerHTML=result
;
}
</script>

</body>
</html>
```

```
<html>
<body>

<p>Click the button to do a global search for characters NOT inside the brackets [h] in a string.</p>

<button onclick="myFunction()">Try it</button>

<p id="demo"></p>

<script>
function myFunction() {
  var str = "Is this all there is?";
  var patt1 = /[^h]/g;
  var result = str.match(patt1);
  document.getElementById("demo").innerHTML = result;
}
</script>

</body>
</html>
```

```
<html>
<body>

<p>Click the button to do a global search for the numbers 1 to 4 in a string.</p>

<button onclick="myFunction()">Try it</button>

<p id="demo"></p>

<script>
function myFunction() {
  var str = "123456789";
  var patt1 = /[1-4]/g;
  var result = str.match(patt1);
  document.getElementById("demo").innerHTML = result;
}
</script>

</body>
</html>
```

```
<html>
<body>

<p>Click the button to do a global search for numbers that are NOT 1 to 4 in a string.</p>

<button onclick="myFunction()">Try it</button>

<p id="demo"></p>

<script>
function myFunction() {
  var str = "123456789";
  var patt1 = /[^1-4]/g;
  var result = str.match(patt1);
  document.getElementById("demo").innerHTML =
result;
}
</script>

</body>
</html>
```

```
<html>
<body>

<p>Click the button to do a global search for any of the specified alternatives (red|green).</p>

<button onclick="myFunction()">Try it</button>

<p id="demo"></p>

<script>
function myFunction() {
  var str = "re, green, red, green, gren, gr, blue, yellow";
  var patt1 = /(red|green)/g;
  var result = str.match(patt1);
  document.getElementById("demo").innerHTML = result;
}
</script>

</body>
</html>
```

Assertions

Assertions include boundaries, which indicate the beginnings and endings of lines and words

^	Matches the beginning of input. If the multiline flag is set to true, also matches immediately after a line break character. For example, /^A/ does not match the "A" in "an A", but does match the first "A" in "An A".
$	Matches the end of input. If the multiline flag is set to true, also matches immediately before a line break character. For example, /t$/ does not match the "t" in "eater", but does match it in "eat".

\b	Matches a word boundary. This is the position where a word character is not followed or preceded by another word-character, such as between a letter and a space. Note that a matched word boundary is not included in the match. In other words, the length of a matched word boundary is zero.

Examples:

/\bm/ matches the "m" in "moon".

/oo\b/ does not match the "oo" in "moon", because "oo" is followed by "n" which is a word character.

/oon\b/ matches the "oon" in "moon", because "oon" is the end of the string, thus not followed by a word character.

/\w\b\w/ will never match anything, because a word character can never be followed by both a non-word and a word character.

\B	Matches a non-word boundary. This is a position where the previous and next character are of the same type: Either both must be words, or both must be non-words, for example between two letters or between two spaces. The beginning and end of a string are considered non-words. Same as the matched word boundary, the matched non-word boundary is also not included in the match. For example, /\Bon/ matches "on" in "at noon", and /ye\B/ matches "ye" in "possibly yesterday".

x(?=y)	**Lookahead assertion:** Matches "x" only if "x" is followed by "y". For example, /Jack(?=Sprat)/ matches "Jack" only if it is followed by "Sprat". /Jack(?=Sprat|Frost)/ matches "Jack" only if it is followed by "Sprat" or "Frost". However, neither "Sprat" nor "Frost" is part of the match results.
x(?!y)	**Negative lookahead assertion**: Matches "x" only if "x" is not followed by "y". For example, /\d+(?!\.)/ matches a number only if it is not followed by a decimal point. /\d+(?!\.)/.exec('3.141') matches "141" but not "3".

(?<=y)x	**Lookbehind assertion**: Matches "x" only if "x" is preceded by "y". For example, /(?<=Jack)Sprat/ matches "Sprat" only if it is preceded by "Jack". /(?<=Jack|Tom)Sprat/ matches "Sprat" only if it is preceded by "Jack" or "Tom". However, neither "Jack" nor "Tom" is part of the match results.
(?<!y)x	Negative lookbehind assertion: Matches "x" only if "x" is not preceded by "y". For example, /(?<!-)\d+/ matches a number only if it is not preceded by a minus sign. /(?<!-)\d+/.exec('3') matches "3". /(?<!-)\d+/.exec('-3') match is not found because the number is preceded by the minus sign.

```
const text = 'A quick fox’;

const regexpLastWord = /\w+$/;console.log(text.match(regexpLastWord));

// expected output: Array ["fox"]


const regexpWords = /\b\w+\b/g;

console.log(text.match(regexpWords));

// expected output: Array ["A", "quick", "fox"]


const regexpFoxQuality = /\w+(?= fox)/;

console.log(text.match(regexpFoxQuality));

// expected output: Array ["quick"]
```

```
// Using Regex boundaries to fix buggy string.
buggyMultiline = `tey, ihe light-greon apple
tangs on ihe greon traa`;

// 1) Use ^ to fix the matching at the begining of the string, and right after newline.
buggyMultiline = buggyMultiline.replace(/^t/gim,'h');
console.log(1, buggyMultiline); // fix 'tey', 'tangs' => 'hey', 'hangs'. Avoid 'traa'.

// 2) Use $ to fix matching at the end of the text.
buggyMultiline = buggyMultiline.replace(/aa$/gim,'ee.');
console.log(2, buggyMultiline); // fix  'traa' => 'tree'.

// 3) Use \b to match characters right on border between a word and a space.
buggyMultiline = buggyMultiline.replace(/\bi/gim,'t');
console.log(3, buggyMultiline); // fix  'ihe' but does not touch 'light'.

// 4) Use \B to match characters inside borders of an entity.
fixedMultiline = buggyMultiline.replace(/\Bo/gim,'e');
console.log(4, fixedMultiline); // fix  'greon' but does not touch 'on'.
```

Use ^ for matching at the beginning of input. In this example, we can get the fruits that start with 'A' by a /^A/ regex. For selecting appropriate fruits we can use the filter method with an arrow function.

```
let fruits = ["Apple", "Watermelon", "Orange", "Avocado", "Strawberry"];

// Select fruits started with 'A' by /^A/ Regex.
// Here '^' control symbol used only in one role: Matching beginning of an input.

let fruitsStartsWithA = fruits.filter(fruit => /^A/.test(fruit));
console.log(fruitsStartsWithA); // [ 'Apple', 'Avocado' ]
```

In the second example ^ is used both for matching at the beginning of input and for creating negated or complemented character class when used within groups.

```
let fruits = ["Apple", "Watermelon", "Orange", "Avocado", "Strawberry"];

// Selecting fruits that dose not start by 'A' by a /^[^A]/ regex.
// In this example, two meanings of '^' control symbol are represented:
// 1) Matching begining of the input
// 2) A negated or complemented character class: [^A]
// That is, it matches anything that is not enclosed in the brackets.

let fruitsStartsWithNotA = fruits.filter(fruit => /^[^A]/.test(fruit));

console.log(fruitsStartsWithNotA); // [ 'Watermelon', 'Orange', 'Strawberry' ]
```

Matching a word boundary

```
let fruitsWithDescription = ["Red apple", "Orange orange", "Green Avocado"];

// Select descriptions that contains 'en' or 'ed' words endings:
let enEdSelection = fruitsWithDescription.filter(descr => /(en|ed)\b/.test(descr));

console.log(enEdSelection); // [ 'Red apple', 'Green Avocado' ]
```

Lookahead assertion

```
// JS Lookahead assertion x(?=y)

let regex = /First(?= test)/g;

console.log('First test'.match(regex)); // [ 'First' ]
console.log('First peach'.match(regex)); // null
console.log('This is a First test in a year.'.match(regex)); // [ 'First' ]
console.log('This is a First peach in a month.'.match(regex)); // null
```

Basic negative lookahead assertion

For example, /\d+(?!\.)/ matches a number only if it is not followed by a decimal point. /\d+(?!\.)/.exec('3.141') matches "141" but not "3.

```
console.log(/\d+(?!\.)/g.exec('3.141')); // [ '141', index: 2, input: '3.141' ]
```

Different meaning of '?!' combination usage in Assertions and Ranges

Different meaning of ?! combination usage in Assertions /x(?!y)/ and Ranges [^?!].

```
let orangeNotLemon = "Do you want to have an orange? Yes, I do not want to have a lemon!";

// Different meaning of '?!' combination usage in Assertions /x(?!y)/ and  Ranges /[^?!]/
let selectNotLemonRegex = /[^?!]+have(?! a lemon)[^?!]+[?!]/gi
console.log(orangeNotLemon.match(selectNotLemonRegex)); // [ 'Do you want to have an orange?' ]

let selectNotOrangeRegex = /[^?!]+have(?! an orange)[^?!]+[?!]/gi
console.log(orangeNotLemon.match(selectNotOrangeRegex)); // [ ' Yes, I do not want to have a lemon!' ]
```

Lookbehind assertion

```
let oranges = ['ripe orange A ', 'green orange B', 'ripe orange C',];

let ripe_oranges = oranges.filter( fruit => fruit.match(/(?<=ripe )orange/));
console.log(ripe_oranges); // [ 'ripe orange A ', 'ripe orange C' ]
```

Regular Expressions- Using String Methods

In JavaScript, regular expressions are often used with the two string methods: search() and replace().

The search() method uses an expression to search for a match, and returns the position of the match.

The replace() method returns a modified string where the pattern is replaced.

Using String search() With a String

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript String Methods</h2>

<p>Search a string for “Chitkara", and display the position of
the match:</p>

<p id="demo"></p>

<script>
var str = "Visit Chitkara!";
var n = str.search(“Chitkara");
document.getElementById("demo").innerHTML = n;
</script>

</body>
</html>
```

The search() method searches a string for a specified value and returns the position of the match:

Using String search() With a Regular Expression

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Regular Expressions</h2>

<p>Search a string for “Chitkara", and display the position of the match:</p>

<p id="demo"></p>

<script>
var str = "Visit Chitkara!";
var n = str.search(/ chitkara /i);
document.getElementById("demo").innerHTML = n;
</script>

</body>
</html>
```

Using String replace() With a String

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript String Methods</h2>
<p>Replace "Microsoft" with " Chitkara" in the paragraph below:</p>
<button onclick="myFunction()">Try it</button>
<p id="demo">Please visit Microsoft!</p>

<script>
function myFunction() {
  var str = document.getElementById("demo").innerHTML;
  var txt = str.replace("Microsoft",“Chitkara");
  document.getElementById("demo").innerHTML = txt;
}
</script>

</body>
</html>
```

Use String replace() With a Regular Expression

```
<html>
<body>

<h2>JavaScript Regular Expressions</h2>

<p>Replace "microsoft" with “Chitkara" in the paragraph below:</p>

<button onclick="myFunction()">Try it</button>

<p id="demo">Please visit Microsoft and Microsoft!</p>

<script>
function myFunction() {
  var str = document.getElementById("demo").innerHTML;
  var txt = str.replace(/microsoft/i,“Chitkara");
  document.getElementById("demo").innerHTML = txt;
}
</script>

</body>
</html>
```

Using test()

The test() method is a RegExp expression method.

It searches a string for a pattern, and returns true or false, depending on the result.

The following example searches a string for the character "e":

```
<html>
<body>

<h2>JavaScript Regular Expressions</h2>

<p>Search for an "e" in the next paragraph:</p>

<p id="p01">The best things in life are free!</p>

<p id="demo"></p>

<script>
text = document.getElementById("p01").innerHTML;
document.getElementById("demo").innerHTML = /e/.test(text);
</script>

</body>
</html>
```

Using exec()

The exec() method is a RegExp expression method.

It searches a string for a specified pattern, and returns the found text as an object.

If no match is found, it returns an empty (null) object.

The following example searches a string for the character "e":

```
<html>
<body>

<h2>JavaScript Regular Expressions</h2>

<p id="demo"></p>

<script>
var obj = /e/.exec("The best things in life are free!");
document.getElementById("demo").innerHTML =
"Found " + obj[0] + " in position " + obj.index + " in the text: " +
obj.input;
</script>

</body>
</html>
```

RegExp Character classes

\d Matches any digit (Arabic numeral). Equivalent to [0-9]. For example, /\d/ or /[0-9]/ matches "2" in "B2 is the suite number".

\D Matches any character that is not a digit (Arabic numeral). Equivalent to [^0-9]. For example, /\D/ or /[^0-9]/ matches "B" in "B2 is the suite number".

\w Matches any alphanumeric character from the basic Latin alphabet, including the underscore. Equivalent to [A-Za-z0-9_]. For example, /\w/ matches "a" in "apple", "5" in "$5.28", "3" in "3D" and "m" in "Émanuel".

\W Matches any character that is not a word character from the basic Latin alphabet. Equivalent to [^A-Za-z0-9_]. For example, /\W/ or /[^A-Za-z0-9_]/ matches "%" in "50%" and "É" in "Émanuel".

\s Matches a single white space character, including space, tab, form feed, line feed, and other Unicode spaces. Equivalent to [\f\n\r\t\v\u00a0\u1680\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]. For example, /\s\w*/ matches " bar" in "foo bar".

\S Matches a single character other than white space. Equivalent to [^ \f\n\r\t\v\u00a0\u1680\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]. For example, /\S\w*/ matches "foo" in "foo bar".

\t Matches a horizontal tab.

\r Matches a carriage return.

\n Matches a linefeed.

\v Matches a vertical tab.

\f Matches a form-feed.

Examples

Looking for a series of digits

```
var randomData = "015 354 8787 687351 3512 8735";
var regexpFourDigits = /\b\d{4}\b/g;
// \b indicates a boundary (i.e. do not start matching in the middle of a word)
// \d{4} indicates a digit, four times
// \b indicates another boundary (i.e. do not end matching in the middle of a word)

console.table(randomData.match(regexpFourDigits));
// ['8787', '3512', '8735']
```

Looking for a word (from the latin alphabet) starting with A

```
var aliceExcerpt = "I’m sure I’m not Ada,’ she said, ‘for her hair goes in such long ringlets, and mine doesn’t go in ringlets at all.";
var regexpWordStartingWithA = /\b[aA]\w+/g;
// \b indicates a boundary (i.e. do not start matching in the middle of a word)
// [aA] indicates the letter a or A
// \w+ indicates any character *from the latin alphabet*, multiple times

console.table(aliceExcerpt.match(regexpWordStartingWithA));
// ['Ada', 'and', 'at', 'all']
```

RegExp Groups and ranges

Types

x|y Matches either "x" or "y". For example, /green|red/ matches "green" in "green apple" and "red" in "red apple".

[xyz]
[a-c] A character class. Matches any one of the enclosed characters. You can specify a range of characters by using a hyphen, but if the hyphen appears as the first or last character enclosed in the square brackets it is taken as a literal hyphen to be included in the character class as a normal character.

For example, [abcd] is the same as [a-d]. They match the "b" in "brisket", and the "c" in "chop".

For example, [abcd-] and [-abcd] match the "b" in "brisket", the "c" in "chop", and the "-" (hyphen) in "non-profit".

[^xyz]

[^a-c]

A negated or complemented character class. That is, it matches anything that is not enclosed in the brackets. You can specify a range of characters by using a hyphen, but if the hyphen appears as the first or last character enclosed in the square brackets it is taken as a literal hyphen to be included in the character class as a normal character. For example, [^abc] is the same as [^a-c]. They initially match "o" in "bacon" and "h" in "chop".

Counting vowels

```
var aliceExcerpt = "There was a long silence after this, and Alice could only hear
whispers now and then.";
var regexpVowels = /[aeiou]/g;

console.log("Number of vowels:", aliceExcerpt.match(regexpVowels).length);
```

Using groups

```
let personList = `First_Name: John, Last_Name: Doe
First_Name: Jane, Last_Name: Smith`;

let regexpNames =  /First_Name: (\w+), Last_Name: (\w+)/mg;
let match = regexpNames.exec(personList);
do {
  console.log(`Hello ${match[1]} ${match[2]}`);
} while((match = regexpNames.exec(personList)) !== null);
```

Using named groups

```
let personList = `First_Name: John, Last_Name: Doe
First_Name: Jane, Last_Name: Smith`;

let regexpNames =  /First_Name: (?<firstname>\w+), Last_Name:
(?<lastname>\w+)/mg;
let match = regexpNames.exec(personList);
do {
  console.log(`Hello ${match.groups.firstname} ${match.groups.lastname}`);
} while((match = regexpNames.exec(personList)) !== null);
```

Quantifiers

Quantifiers indicate numbers of characters or expressions to match.

x*	Matches the preceding item "x" 0 or more times. For example, /bo*/ matches "boooo" in "A ghost booooed" and "b" in "A bird warbled", but nothing in "A goat grunted".
x+	Matches the preceding item "x" 1 or more times. Equivalent to {1,}. For example, /a+/ matches the "a" in "candy" and all the "a"'s in "caaaaaaandy".
x?	Matches the preceding item "x" 0 or 1 times. For example, /e?le?/ matches the "el" in "angel" and the "le" in "angle."
x{n}	Where "n" is a positive integer, matches exactly "n" occurrences of the preceding item "x". For example, /a{2}/ doesn't match the "a" in "candy", but it matches all of the "a"'s in "caandy", and the first two "a"'s in "caaandy".
x{n,}	Where "n" is a positive integer, matches at least "n" occurrences of the preceding item "x". For example, /a{2,}/ doesn't match the "a" in "candy", but matches all of the a's in "caandy" and in "caaaaaaandy".

x{n,m} Where "n" is 0 or a positive integer, "m" is a positive integer, and m > n, matches at least "n" and at most "m" occurrences of the preceding item "x". For example, /a{1,3}/ matches nothing in "cndy", the "a" in "candy", the two "a"'s in "caandy", and the first three "a"'s in "caaaaaaandy". Notice that when matching "caaaaaaandy", the match is "aaa", even though the original string had more "a"s in it.

x*?

x+?

x??

x{n}?

x{n,}?

x{n,m}? By default quantifiers like * and + are "greedy", meaning that they try to match as much of the string as possible. The ? character after the quantifier makes the quantifier "non-greedy": meaning that it will stop as soon as it finds a match. For example, given a string like "some <foo> <bar> new </bar> </foo> thing":

/<.*>/ will match "<foo> <bar> new </bar> </foo>"

/<.*?>/ will match "<foo>"

Repeated pattern

```
var wordEndingWithAs = /\w+a+\b/;
var delicateMessage = "This is Spartaaaaaaa";

console.table(delicateMessage.match(wordEndingWithAs)); // [ "Spartaaaaaaa"
]
```

Counting characters

```
var singleLetterWord = /\b\w\b/g;
var notSoLongWord = /\b\w{1,6}\b/g;
var loooongWord = /\b\w{13,}\b/g;

var sentence = "Why do I have to learn multiplication table?";

console.table(sentence.match(singleLetterWord)); // ["I"]
console.table(sentence.match(notSoLongWord));    // [ "Why", "do", "I", "have",
"to", "learn", "table" ]
console.table(sentence.match(loooongWord));      // ["multiplication"]
```

Optional character

```
var britishText = "He asked his neighbour a favour.";
var americanText = "He asked his neighbor a favor.";

var regexpEnding = /\w+ou?r/g;
// \w+ One or several letters
// o   followed by an "o",
// u?  optionally followed by a "u"
// r   followed by an "r"

console.table(britishText.match(regexpEnding));
// ["neighbour", "favour"]

console.table(americanText.match(regexpEnding));
// ["neighbor", "favor"]
```

Greedy versus non-greedy

```
var text = "I must be getting somewhere near the centre of the earth.";
var greedyRegexp = /[\w ]+/;
// [\w ]      a letter of the latin alphabet or a whitespace
//      +     one or several times

console.log(text.match(greedyRegexp)[0]);
// "I must be getting somewhere near the centre of the earth"
// almost all of the text matches (leaves out the dot character)

var nonGreedyRegexp = /[\w ]+?/; // Notice the question mark
console.log(text.match(nonGreedyRegexp));
// "I"
// The match is the smallest one possible
```