# **JavaScript Regular Expressions**

* A regular expression is a sequence of characters that forms a search pattern.
* The search pattern can be used for text search and text replace operations.

## **What Is a Regular Expression?**

* A regular expression is a sequence of characters that forms a **search pattern**.
* When you search for data in a text, you can use this search pattern to describe what you are searching for.
* A regular expression can be a single character, or a more complicated pattern.
* Regular expressions can be used to perform all types of **text search** and **text replace** operations.

## **Syntax**

/pattern/modifiers;

### **Example**

/CSE/i;

Example explained:

**/CSE/i**  is a regular expression.

CSE  is a pattern (to be used in a search).

**i**  is a modifier (modifies the search to be case-insensitive

## **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

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

### **Example**

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript String Methods</h1>

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

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

<script>

let text = "Visit CSE!";

let n = text.search("CSE");

document.getElementById("demo").innerHTML = n;

</script>

</body>

</html>

**The result:**

# JavaScript String Methods

Search a string for "CSE", and display the position of the match:

6

## Using String search() With a Regular Expression

### **Example**

Use a regular expression to do a case-insensitive search for "w3schools" in a string:

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Regular Expressions</h2>

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

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

<script>

let text = "Visit Cse";

let n = text.search(/cse/i);

document.getElementById("demo").innerHTML = n;

</script>

</body>

</html>

**The result:**

## JavaScript Regular Expressions

Search a string for "cse", and display the position of the match:

6

## Using String replace() With a String

The replace() method replaces a specified value with another value in a string:

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript String Methods</h1>

<p>Replace "Microsoft" with "Ubuntu" in the paragraph below:</p>

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

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

<script>

function myFunction() {

 let text = document.getElementById("demo").innerHTML;

 document.getElementById("demo").innerHTML =

 text.replace("Microsoft","Ubuntu");

}

</script>

</body>

</html>

The result:

# JavaScript String Methods

Replace "Microsoft" with "Ubuntu" in the paragraph below:

Please visit Microsoft!

Try it

## Use String replace() With a Regular Expression

### **Example**

Use a case insensitive regular expression to replace Microsoft with Ubuntu in a string:

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript String Methods</h2>

<p>Replace "Microsoft" with "Ubuntu" in the paragraph below:</p>

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

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

<script>

function myFunction() {

 let text = document.getElementById("demo").innerHTML;

 document.getElementById("demo").innerHTML =

 text.replace(/microsoft/i, "Ubuntu");

}

</script>

</body>

</html>

**The result :**

## JavaScript String Methods

Replace "Microsoft" with "Ubuntu" in the paragraph below:

Try it

Please visit Microsoft!

## Did You Notice?

Regular expression arguments (instead of string arguments) can be used in the methods above.
Regular expressions can make your search much more powerful (case insensitive for example).

## Regular Expression Modifiers

**Modifiers** can be used to perform case-insensitive more global searches:

|  |  |  |
| --- | --- | --- |
| Modifier | Description | Try it |
| i | Perform case-insensitive matching | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_i" \t "https://www.w3schools.com/js/_blank) |
| g | Perform a global match (find all) | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_g" \t "https://www.w3schools.com/js/_blank) |
| m | Perform multiline matching | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_m" \t "https://www.w3schools.com/js/_blank) |
| d | Perform start and end matching (New in [ES2022](https://www.w3schools.com/js/js_2022.asp)) | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_inices" \t "https://www.w3schools.com/js/_blank) |

## Regular Expression Patterns

**Brackets** are used to find a range of characters:

|  |  |  |
| --- | --- | --- |
| Expression | Description | Try it |
| [abc] | Find any of the characters between the brackets | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_abc" \t "https://www.w3schools.com/js/_blank) |
| [0-9] | Find any of the digits between the brackets | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_0-9" \t "https://www.w3schools.com/js/_blank) |
| (x|y) | Find any of the alternatives separated with | | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_xy" \t "https://www.w3schools.com/js/_blank) |

**Metacharacters** are characters with a special meaning:

|  |  |  |
| --- | --- | --- |
| Metacharacter | Description | Try it |
| \d | Find a digit | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_d" \t "https://www.w3schools.com/js/_blank) |
| \s | Find a whitespace character | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_s" \t "https://www.w3schools.com/js/_blank) |
| \b | Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_b" \t "https://www.w3schools.com/js/_blank)[Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_b2" \t "https://www.w3schools.com/js/_blank) |
| \uxxxx | Find the Unicode character specified by the hexadecimal number xxxx | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_ux" \t "https://www.w3schools.com/js/_blank) |

**Quantifiers** define quantities:

|  |  |  |
| --- | --- | --- |
| Quantifier | Description | Try it |
| n+ | Matches any string that contains at least one n | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_n1" \t "https://www.w3schools.com/js/_blank) |
| n\* | Matches any string that contains zero or more occurrences of n | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_n2" \t "https://www.w3schools.com/js/_blank) |
| n? | Matches any string that contains zero or one occurrences of n | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_n3" \t "https://www.w3schools.com/js/_blank) |

## **Using the RegExp Object**

* In JavaScript, the RegExp object is a regular expression object with predefined properties and methods.

## In JavaScript, there are two main ways to create regular expression objects: using the RegExp constructor function or using the regular expression literal notation.

## **Using the RegExp Constructor Function:**

## You can create a regular expression object using the RegExp constructor function. This method is useful when the regular expression pattern is dynamic and needs to be constructed at runtime.

## **Code:**

## var regex = new RegExp('pattern');

## You can also include flags as a second argument to the constructor to modify the behavior of the regular expression:

## **Code:**

## var regex = new RegExp('pattern', 'g');

## In this example, 'g' is the flag for global matching.

## **Using Regular Expression Literal Notation:**

## Regular expression literal notation is a shorthand way of creating regular expression objects directly in your code. This method is useful when the regular expression pattern is known and static.

## **Code:**

## var regex = /pattern/;

## You can also include flags directly within the literal notation:

## **Code:**

## var regex = /pattern/g;

## In this example, /g is the flag for global matching.

## Both methods achieve the same result, but the choice between them depends on your specific requirements and coding style. If the regular expression pattern is static, the literal notation is often preferred for its readability and conciseness. If the pattern is dynamic and needs to be constructed at runtime, using the RegExp constructor function is more appropriate.

## **Using test()**

The test() method is a RegExp expression object.

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":

### **Example**

<!DOCTYPE html>

<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>

let text = document.getElementById("p01").innerHTML;

const pattern = /e/;

document.getElementById("demo").innerHTML = pattern.test(text);

</script>

</body>

</html>

Since there is an "e" in the string, the output of the code above will be:

**The result:**

## JavaScript Regular Expressions

Search for an "e" in the next paragraph:

The best things in life are free!

true

You don't have to put the regular expression in a variable first. The two lines above can be shortened to one:

/e/.test("The best things in life are free!");

## **Using exec()**

The exec() method is a RegExp expression object.

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":

**Example**

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Regular Expressions</h2>

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

<script>

const 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>

**The result:**

JavaScript Regular Expressions

Found e in position 2 in the text: The best things in life are free!

Limitations of regular expressions

**Limitations Of RegExp:**

While regular expressions are powerful tools for pattern matching and text manipulation, they do have some limitations:

**1.Complexity with Nested Structures:**

Regular expressions become increasingly complex and difficult to read when dealing with nested or recursive patterns. Matching nested HTML tags, for example, can quickly become unwieldy.

**2.Limited Context Awareness:**

Regular expressions operate on a character-by-character basis and lack the ability to understand the context or semantics of the text they are matching. This can lead to false positives or negatives in certain cases.

**3.Performance Issues:**

Regular expressions can be inefficient, especially for large or complex patterns, leading to slow execution times. Catastrophic backtracking, a situation where the regex engine explores an exponentially large number of permutations trying to find a match, can occur in certain scenarios.

**4.Difficulty with Some Text Parsing Tasks:**

Regular expressions are not well-suited for parsing tasks that involve balancing or counting, such as matching balanced parentheses or parsing recursive structures. While possible with advanced techniques, such tasks are often better handled by parsers or specialized tools.

**5.Limited Language Support:**

Different implementations of regular expressions across programming languages and libraries may have varying levels of support for certain features or syntax. This can lead to portability issues when moving regex patterns between different environments.

**6.Readability and Maintainability:**

Complex regular expressions can be challenging to understand and maintain, especially for developers who are not familiar with the intricacies of regex syntax. This can lead to errors and difficulties in debugging and maintaining code over time.

**7.Difficulty in Handling Unicode and Multilingual Text:**

While modern regex engines have improved support for Unicode characters, handling multilingual text and complex Unicode properties can still be challenging with regular expressions.