



Co-funded by the
Erasmus+ Programme
of the European Union



3.1.:

Code4SP Training Material Package

WP3:

Code4SP Training
Materials - JavaScript

Prepared by:



Project Information

Project Acronym: Code4SP

Project Title: Coding for Social Promotion

Project Reference: 621417-EPP-1-2020-1-PT-EPPKA3-IPI-SOC-IN

Project website: www.code4sp.eu

Authoring Partner: Various Authors

Document Version: 2

Date of Preparation: 03/03/2022

Document History			
Date	Version	Author	Description
03/03/2022	1	Various Authors	Draft
22/03/2022	2	Various Authors	Revision

Table of contents

Topic Information:	4
5.1. JavaScript Basic.....	7
5.2. JavaScript & DOM.....	82
5.3. JavaScript & BOM.....	111
5.4. JavaScript Advanced	131

Topic:

5. JavaScript

Prerequisites:

To learn JavaScript, learners must know the basics of HTML and CSS. For a working knowledge of JavaScript and most web-based projects, this knowledge will be sufficient. For more advanced projects and skills, it is recommended to know basic OOP concepts and an OOP based programming language (like Java).

Workload:

15 hours.

Description:

JavaScript is a programming language that allows the developer to perform changes on the content of a web page in a dynamic way. JavaScript is among the most powerful and flexible programming languages of the web.

This JavaScript topic covers all the fundamental programming concepts, including data types, operators, creating and using variables, generating outputs, structuring the code to make decisions in programs or to loop over the same block of code multiple times, creating and manipulating strings and arrays, defining and calling functions, etc.

Once learners are comfortable with the basics, they will move on to next level that explains the idea of objects, the Document Object Model (DOM) and Browser Object Model (BOM), as well as how to make use of the native JavaScript objects like Date, Math, etc., and perform type conversions.

Moreover, some advanced concepts like event listeners, event propagation, borrowing methods from other objects, hoisting behavior of JavaScript, encoding and decoding JSON data, as well as detailed overview of new features introduced in ECMAScript 6 (or ES6) will be explored.

Learning outcomes:

Learners will understand how JavaScript allows the developer to change content dynamically and modify the HTML or CSS information on the client side, when the webpage is being shown to the user. Therefore, learners will study the structures of JavaScript code; will get to know how to change HTML/CSS code on the loading of the page and also will learn how to create a function and attach it to an event.

Material required:

- Computer or laptop
- Internet connection
- Text editor (online or offline): [Sublime Text](#)/[Brackets](#)/[W3Schools online editor](#)

Lesson Scenario:

The total time for this topic is 15 hours, and it will be up to the trainer/coach to decide how much time to dedicate to teaching each subtopic. In order to make the most of all the time available, we propose the use of the training materials produced by the project (PPT presentations), which were designed with an effective use of time in mind. These presentations are composed of the following elements:

- Development of the subtopic and main ideas to retain;
- Proposed Activities/Exercises.

That said, if the trainer/coach follows the logical sequence of the PPTs, he/she will certainly be able to complete the session within the stipulated time limit. These presentations can also be made available to learners for individual study.

Subtopics:

- JavaScript Basic
- JavaScript & DOM
- JavaScript & BOM
- JavaScript Advanced
- JavaScript Reference

Additional resources:

- JavaScript tutorial: [w3schools](https://www.w3schools.com/)
- Online Course on JavaScript: [CodeAcademy](https://www.codecademy.com/)

5.1. JavaScript Basic

JavaScript (JS) is the most widespread *client-side* scripting language (client-side scripting is associated to scripts running within a web browser). JS is intended to add interactivity and dynamic effects to the web pages by manipulating the content returned from a web server.

JavaScript was first developed as LiveScript by Netscape's computer programmer Brendan Eich in 1995. It was later renamed to JavaScript, and became an ECMA (European Computer Manufacturers Association) standard in 1997. Nowadays, JavaScript is the standard client-side scripting language for web-based applications, and it is supported by nearly all web browsers available (Chrome, Firefox, Safari, etc.).

JavaScript is an object-oriented language, and it also has some similarities in syntax to Java programming language, even though it is not related to Java at all.

JavaScript can be used for various purposes:

- Modify the content of a web page by adding or removing elements;
- Change the style and position of the elements on a web page;
- Monitor events like mouse click, hover, etc. and react to it;
- Make and control transitions and animations;
- Produce alert pop-ups to display info or warning messages to the user;
- Complete operations based on user inputs and display the results;
- Validate user inputs before submitting them to the server;
- And many other interesting purposes to be checked later.

Getting started with JavaScript

From this point, learners will understand how simple it can be to add interactivity to a web page by using JavaScript.

Characteristically, there are 3 ways of adding JS to a webpage:

- Embedding the JavaScript code between a `<script>` and a `</script>` tag;
- Creating an external JavaScript file with the `.js` extension and then load it within the page through the `src` attribute of the `<script>` tag.
- Placing the JavaScript code directly inside an HTML tag using the special tag attributes such as `onclick`, `onmouseover`, `onkeypress`, `onload`, etc.

a) Embedding the JavaScript code between a `<script>` and a `</script>` tag;

The JavaScript code can be embedded directly within a web page by placing it between the `<script>` and `</script>` tags. The `<script>` tag indicates the browser that the contained statements are to be interpreted as executable script and not HTML, as checked in the following example:

<pre>1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="UTF-8"> 5 <title>Embedding JavaScript</title> 6 </head> 7 <body> 8 <script> 9 var greet = "Hello World!"; 10 document.write(greet); // Prints: Hello World! 11 </script> 12 </body> 13 </html></pre>	Hello World!
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------

Figure 1 – Embedding JS code between a `<script>` and a `</script>` tag (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

b) Creating an external JavaScript file with the `.js` extension and then load it within the page through the `src` attribute of the `<script>` tag.

A JavaScript code can be placed as well into a separate file with a .js extension, being then called in that file in the same document through the src attribute of the <script> tag, as follows:

```
<script src="js/hello.js"></script>
```

This is especially valuable if the programmer wants the same scripts available to multiple documents. Following this procedure, he/she will avoid repeating the same task over and over again, and it makes his/her website much simpler to maintain.

Forward to the statement above, a JavaScript file named "hello.js" will be created and the following code shall be inserted in it:

```
1 // A function to display a message
2 function sayHello() {
3     alert("Hello World!");
4 }
5
6 // Call function on click of the button
7 document.getElementById("myBtn").onclick = sayHello;
```

Figure 2 – Creating a file and calling a function on JS (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

At this point, the programmer may call the above JS file within a webpage by using the <script> tag, as seen in the image below:



Figure 3 – Creating a file and calling a function on JS (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

c) Placing the JavaScript code directly inside an HTML tag using the special tag attributes such as onclick, onmouseover, onkeypress, onload, etc.

JavaScript code may be introduced inline by inserting it directly inside the HTML tag by means of the special tag attributes such as onclick, onmouseover, onkeypress, onload, etc.

Nonetheless, it is not advisable to place large amount of JavaScript code inline as it disorders up HTML with JavaScript and makes JS code hard to maintain. *Figure 4* shows an example (in this case, an alert message is shown upon clicking on the button element):



Figure 4 – Placing the JS code inline (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

The `<script>` element can be positioned in the `<head>` or `<body>` section of an HTML document. However, scripts should be preferably positioned at the end of the body section, before the closing `</body>` tag. This procedure will enable web pages to load faster, since it avoids obstruction of initial page rendering. Each `<script>` tag blocks the page rendering process until it has fully downloaded and executed the JavaScript code, so placing them in the head section (i.e. `<head>` element) of the document without any valid reason will significantly impact the performance of a website.

There are differences between Client-side and Server-side Scripting. Client-side scripting languages (e.g., JavaScript or VBScript are understood and executed by the web browser, in opposite of server-side scripting languages (e.g., PHP, ASP, Java, Python, Ruby, etc.), which run on the web server and their output is sent back to the web browser in HTML format.

Client-side scripting has many advantages comparing to traditional server-side scripting. For instance, JavaScript can be used to check if the user has entered invalid data in form fields and show notifications for input errors consequently in real-time before submitting the form to the web-server for final data validation and further processing in order to avoid needless network bandwidth usages and the misuse of server system resources.

Likewise, response from a server-side script is slower as compared to a client-side script, as server-side scripts are processed on a remote computer (not on a local one).

JavaScript Syntax

It is time to learn how to write the JS code.

Firstly, it is important to understand the JavaScript Syntax.

The syntax of JS is the **set of rules** that comprise a well-structured JavaScript programme. JS involves statements that are placed within the `<script> </script>` HTML tags in a web page, or within the external JavaScript file having .js extension.

Figure 5 below presents an example of a JS statement:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>Example of JavaScript Statements</title>
6 </head>
7 <body>
8   <script>
9     var x = 5;
10    var y = 10;
11    var sum = x + y;
12    document.write(sum); // Prints variable value
13  </script>
14 </body>
15 </html>
    
```

Figure 5 – Example of a JavaScript statement (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Learners should state that JavaScript is **case-sensitive**. So, variables, language keywords, function names and other identifiers must be typed consistently in terms of letters capitalisation. For instance, the variable myVar must be typed this way (not “MYVAR”, “myvar”, etc.). This applies for **all cases**.

Commonly to the previous topics, JavaScript also provides the possibility of writing comments throughout the coding lines. Comments are inserted mainly because they provide extra information to the source code, but also because it can help programmers on understanding their codes after some time, teamworking, etc.

It is possible to add both single-line as well as multi-line comments on JavaScript. Single-line comments start with a double forward slash (//), followed by the comment text:

```

1 // This is my first JavaScript program
2 document.write("Hello World!");
    
```

Figure 6 – Single-line comment on JavaScript (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

For a multi-line comment, a slash and an asterisk (/*) are the starting point, ending with an asterisk and a slash (*/):

```
1  /* This is my first program
2  in JavaScript */
3  document.write("Hello World!");
```

Figure 7 – Multi-line comment on JavaScript (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

JavaScript Variables

For storing data in JavaScript, programmers create variables. They are key for all the programming languages, and are used to store data, for instance by string of text, numbers, or other element(s). Whenever the programmer needs, he/she can set, update and retrieve data or value stored in the variables. Variables can be understood as symbolic names for values.

A variable can be created by using the var keyword, in which the assignment operator (“=”) is used to allocate value to a variable, as follows: `var varName = value`

```
1  var name = "Peter Parker";
2  var age = 21;
3  var isMarried = false;
```

Figure 8 – Creating a variable (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

In this example, three variables have been created:

Number 1 – string value

Number 2 – number

Number 3 – Boolean value

In JS, some variables are created with the purpose of holding values such as user inputs. That being said, they can be declared without having any initial values associated, as follows:

```
1 // Declaring Variable
2 var userName;
3
4 // Assigning value
5 userName = "Clark Kent";
```

Figure 9 – Creating a variable without having any initial values assigned to them (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Moreover, the programmer may also declare multiple variables and, in a single statement, set their initial values. Each variable is detached by commas, as follows:

```
1 // Declaring multiple Variables
2 var name = "Peter Parker", age = 21, isMarried = false;
3
4 /* Longer declarations can be written to span
5 multiple lines to improve the readability */
6 var name = "Peter Parker",
7 age = 21,
8 isMarried = false;
```

Figure 10 – Declaring multiple variables (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

The latest revision of JavaScript (ECMAScript 2015 or ES6) introduces two new keywords for declaring variables: let and const.

The const keyword works the same way as let. However, variables declared using const cannot be reassigned later in the code, as follows:

```

1 // Declaring variables
2 let name = "Harry Potter";
3 let age = 11;
4 let isStudent = true;
5
6 // Declaring constant
7 const PI = 3.14;
8 console.log(PI); // 3.14
9
10 // Trying to reassign
11 PI = 10; // error

```

Figure 11 – Declaring multiple variables (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

In opposite to var keyword, which declare function-scoped variables, both let and const keywords declare variables, scoped at block-level ({}). Block scoping means that a new scope is created between a pair of curly brackets.

JavaScript variables have **specific rules** for being named:

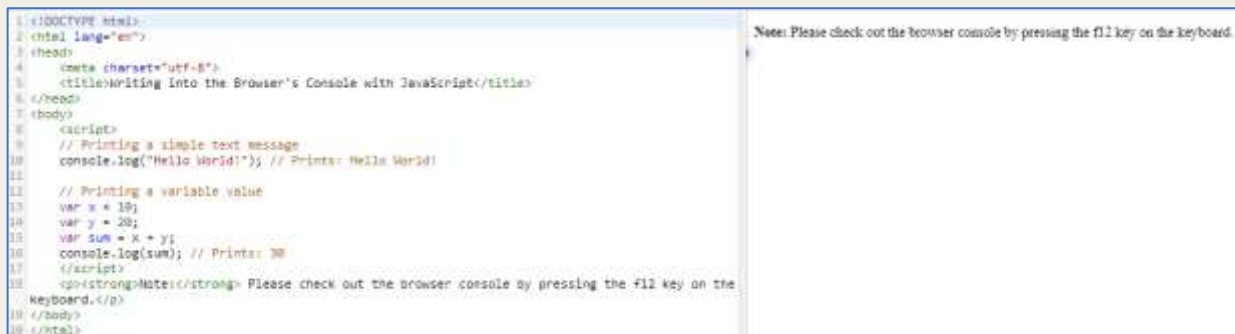
- A variable name must start with a letter, underscore (_), or dollar sign (\$).
- A variable name cannot start with a number.
- A variable name can only comprise alpha-numeric characters (A-Z, 0-9) and underscores.
- A variable name cannot comprehend spaces.
- A variable name cannot be a JavaScript keyword or a JavaScript reserved word.

JavaScript Generating Output

There are certain situations in which programmers may need to create outputs from the JS code, e.g., see the variable's value, write a message to browser console, etc. In JavaScript, there are some different ways of creating output including writing output to

the browser window or browser console, displaying output in dialog boxes, writing output into an HTML element, etc.

It is not difficult to output a message or to write data to the browser's console (it can be accessed by clicking F12). For that purpose, the `console.log()` , a powerful yet simple method, should be applied.



```

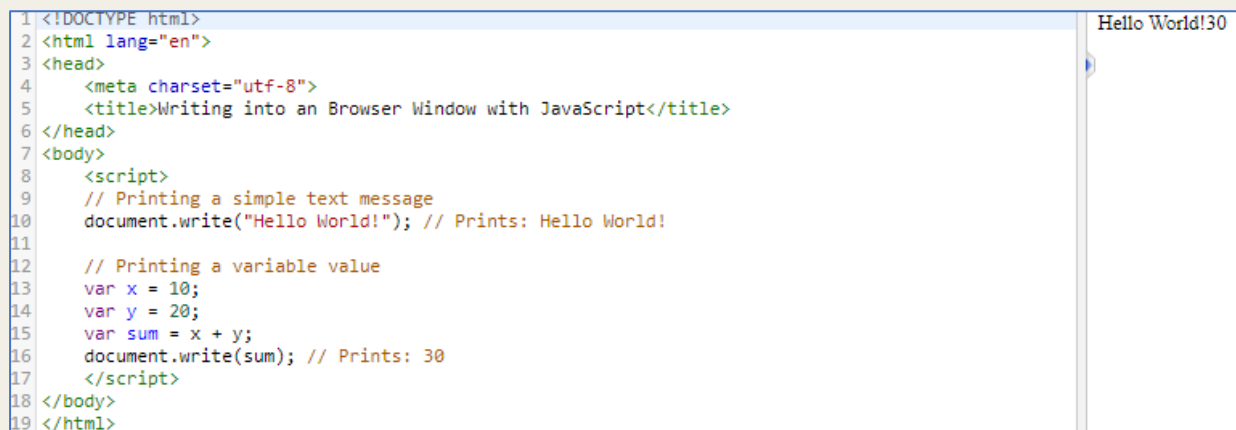
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>Writing into the Browser's Console with JavaScript</title>
6 </head>
7 <body>
8   <script>
9     // Printing a simple text message
10    console.log("Hello World!"); // Prints: Hello World!
11
12    // Printing a variable value
13    var x = 10;
14    var y = 20;
15    var sum = x + y;
16    console.log(sum); // Prints: 30
17  </script>
18  <p><strong>Note:</strong> Please check out the browser console by pressing the F12 key on the
    keyboard.</p>
19 </body>
20 </html>

```

Note: Please check out the browser console by pressing the F12 key on the keyboard.

Figure 12 – Generating outputs by using `console.log` (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

To write the content to the current document only while that document is being deconstructed, the `document.write()` method can be used.



```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>Writing into an Browser Window with JavaScript</title>
6 </head>
7 <body>
8   <script>
9     // Printing a simple text message
10    document.write("Hello World!"); // Prints: Hello World!
11
12    // Printing a variable value
13    var x = 10;
14    var y = 20;
15    var sum = x + y;
16    document.write(sum); // Prints: 30
17  </script>
18 </body>
19 </html>

```

Hello World!30

Figure 13 – Generating outputs by using `document.write` (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

If the document.write() method is used after the page has been loaded, it will overwrite all the existing content in that document, as follows [in this link](#).

Alert dialog boxes can also be added to display the message or output data to the user. To create an alert dialog, the alert() method is used, as follows:

```

1 // Displaying a simple text message
2 alert("Hello World!"); // Outputs: Hello World!
3
4 // Displaying a variable value
5 var x = 10;
6 var y = 20;
7 var sum = x + y;
8 alert(sum); // Outputs: 30

```

Figure 14 – Generating alert dialog boxes (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Outputs can be inserted or written inside an HTML element using the innerHTML property. Nonetheless, the programmer should select the element before writing the output, using the getElementById() method.

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>Writing into an HTML Element with JavaScript</title> 6 </head> 7 <body> 8 <p id="greet"></p> 9 <p id="result"></p> 10 11 <script> 12 // Writing text string inside an element 13 document.getElementById("greet").innerHTML = "Hello World!"; 14 15 // Writing a variable value inside an element 16 var x = 10; 17 var y = 20; 18 var sum = x + y; 19 document.getElementById("result").innerHTML = sum; 20 </script> 21 </body> 22 </html> </pre>	<p>Hello World!</p> <p>30</p>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------

Figure 15 – Using the getElementById () method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

JavaScript Data Types

Data types essentially stipulate what kind of data can be stored and manipulated within a program. There are six basic data types in JS, which can be divided into three main categories:

- Primitive (or primary) – String, Number, and Boolean are examples of primitive data types, which can hold only one value at a time;
- Composite (or reference) – Object, Array, and Function (which are all types of objects) are composite data types. These can hold collections of values and more complex entities; and
- Special data types – Undefined and Null are special data types.

The String data type

It is used to embody textual data (for instance, sequences of characters). Strings are created using single or double quotes surrounding one or more characters:

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript String Data Type</title> 6 </head> 7 <body> 8 <script> 9 // Creating variables 10 var a = 'Hi there!'; // using single quotes 11 var b = "Hi there!"; // using double quotes 12 13 // Printing variable values 14 document.write(a + "
"); 15 document.write(b); 16 </script> 17 </body> 18 </html> </pre>	<p>Hi there! Hi there!</p>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------

Figure 16 – The String data type (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

It should be noted that quotes can be included inside the string, but they should not match the enclosing quotes, as follows [on this example](#).

The Number Data Type

The number data type is useful for exhibiting positive or negative numbers with or without decimal place, or numbers written using exponential notation, for instance: 1.5×10^{-4} (equivalent to 1.5×10^{-4}).

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Number Data Type</title> 6 </head> 7 <body> 8 <script> 9 // Creating variables 10 var a = 25; 11 var b = 80.5; 12 var c = 4.25e+6; 13 var d = 4.25e-6; 14 15 // Printing variable values 16 document.write(a + "
"); 17 document.write(b + "
"); 18 document.write(c + "
"); 19 document.write(d); 20 </script> 21 </body> 22 </html> </pre>	<pre> 25 80.5 4250000 0.00000425 </pre>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------

Figure 17 – The String data type (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

The Number Data Type also comprises some special values which are: Infinity, -Infinity and NaN. Infinity represents the mathematical infinity (∞), which is greater than any number. Infinity is the result of dividing a nonzero number by 0, as could be checked on Figure 18:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Infinity</title>
6 </head>
7 <body>
8   <script>
9     document.write(16 / 0);
10    document.write("<br>");
11    document.write(-16 / 0);
12    document.write("<br>");
13    document.write(16 / -0);
14  </script>
15 </body>
16 </html>

```

Infinity
-Infinity
-Infinity

Figure 18 – The Number Data Type (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

However, NaN represents a special Not-a-Number value. It is a result of an invalid or an undefined mathematical operation, for example, taking the square root of -1 or dividing 0 by 0, etc.

The Boolean Data Type

Two values can be hold in this data type: true or false. It is classically used to stock values like yes (true) or no (false), on (true) or off (false), etc. as follows:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Boolean Data Type</title>
6 </head>
7 <body>
8   <script>
9     // Creating variables
10    var isReading = true; // yes, I'm reading
11    var isSleeping = false; // no, I'm not sleeping
12
13    // Printing variable values
14    document.write(isReading + "<br>");
15    document.write(isSleeping);
16  </script>
17 </body>
18 </html>

```

true
false

Figure 19 – The Boolean Data Type (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Boolean values likewise come as an outcome of comparisons in a program. [This](#) example relates two variables and displays the result in an alert dialog box.

The Undefined Data Type

The undefined data type can only take one value – the special value undefined. If a variable has been declared, but has not been assigned a value, the value shall be declared undefined.

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Undefined Data Type</title>
6 </head>
7 <body>
8   <script>
9     // Creating variables
10    var a;
11    var b = "Hello World!"
12
13    // Printing variable values
14    document.write(a + "<br>");
15    document.write(b);
16  </script>
17 </body>
18 </html>
```

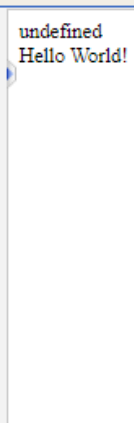


Figure 20 – The Undefined Data Type (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

The Null Data Type

This is one more special data type that can have only one value – the null value. A null value means that simply there is no value. It is not equivalent to an empty string ("") or zero, it is purely nothing.

A variable can be clearly emptied of its current contents by assigning it the **null value**.

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Null Data Type</title>
6 </head>
7 <body>
8   <script>
9     var a = null;
10    document.write(a + "<br>"); // Print: null
11
12    var b = "Hello World!"
13    document.write(b + "<br>"); // Print: Hello World!
14
15    b = null;
16    document.write(b) // Print: null
17  </script>
18 </body>
19 </html>

```

null
Hello World!
null

Figure 21 – The Null Data Type (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

The Object Data Type

The object is a multifaceted data type that permits to store data collections.

An object contains properties, defined as a key-value pair. A property key (name) is always a string, but the value can be any data type (strings, numbers, booleans, or complex data types like arrays, function and other objects). The simplest way to create an object in JavaScript is shown below:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Object Data Type</title>
6 </head>
7 <body>
8   <script>
9     var emptyObject = {};
10    var person = {"name": "Clark", "surname": "Kent", "age": "30"};
11
12    // For better reading
13    var car = {
14      "model": "BMW X3",
15      "color": "white",
16      "doors": 5
17    };
18
19    // Print variables values in browser's console:
20    console.log(person);
21    console.log(car);
22  </script>
23  <p><strong>Note:</strong> Check out the browser console by pressing the F12 key on the
  Keyboard.</p>
24 </body>
25 </html>

```

Note: Check out the browser console by pressing the F12 key on the keyboard.

Figure 22 – The simplest way for creating an object in JS (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

The quotes around property name may be omitted if the name is a valid JS name. This means quotes are required around "first-name" but are non-compulsory around firstname. Thus, the car object in *Figure 22* can also be written as follows:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Object Properties Names without Quotes</title>
6 </head>
7 <body>
8   <script>
9     var car = {
10      model: "BMW X3",
11      color: "white",
12      doors: 5
13    }
14
15    // Print variable value in browser's console
16    console.log(car);
17  </script>
18  <p><strong>Note:</strong> Check out the browser console by pressing the F12 key on the
19  keyboard.</p>
20 </body>
21 </html>

```

Figure 23 – Rewriting the car object from Fig. 22 (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

The Array Data Type

An array is a type of object used for packing multiple values in single variable. Each value (also known as ‘element’) in an array has a numeric position, known as its index, and it may comprise data of any data type-numbers, strings, booleans, functions, objects, and even other arrays. The array index starts from 0 (zero), so that the first array element is arr[0] not arr[1].

The simplest way to make an array is by stipulating the array elements as a comma-separated list enclosed by square brackets, as follows:

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Array Data Type</title> 6 </head> 7 <body> 8 <script> 9 // Creating arrays 10 var colors = ["Red", "Yellow", "Green", "Orange"]; 11 var cities = ["London", "Paris", "New York"]; 12 13 // Printing array values 14 document.write(colors[0] + "
"); // Output: Red 15 document.write(cities[2]); // Output: New York 16 </script> 17 </body> 18 </html> </pre>	Red New York
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------

Figure 24 – Rewriting the car object from Fig. 22 (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

The Function Data Type

The function is a callable object that makes a block of code. Functions are objects, thus it is possible to assign them to variables, as follows:

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Function Data Type</title> 6 </head> 7 <body> 8 <script> 9 var greeting = function(){ 10 return "Hello World!"; 11 } 12 13 // Check the type of greeting variable 14 document.write(typeof greeting) // Output: function 15 document.write("
"); 16 document.write(greeting()); // Output: Hello World! 17 </script> 18 </body> 19 </html> </pre>	function Hello World!
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------

Figure 25 – The Function Data Type (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Actually, functions can be used anywhere, and other values can be used. They can be stored in variables, objects, and arrays. Functions can be approved as arguments to other functions, and they can be returned from functions.

The typeof Operator

The typeof operator can be used to realize what type of data a variable covers. It can be used with or without parentheses (typeof(x) or typeof x).

The typeof operator is mostly beneficial to process the values of different types in a different way. However, the programmer should be cautious, as it may produce unforeseen results in some cases:

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript typeof Operator</title> 6 </head> 7 <body> 8 <script> 9 // Numbers 10 document.write(typeof 15 + "
"); // Prints: "number" 11 document.write(typeof 42.7 + "
"); // Prints: "number" 12 document.write(typeof 2.5e-4 + "
"); // Prints: "number" 13 document.write(typeof Infinity + "
"); // Prints: "number" 14 document.write(typeof NaN + "
"); // Prints: "number". Despite being "Not-A-Number" 15 16 // Strings 17 document.write(typeof '' + "
"); // Prints: "string" 18 document.write(typeof 'hello' + "
"); // Prints: "string" 19 document.write(typeof '12' + "
"); // Prints: "string". Number within quotes is document.write(typeof string 20 21 // Booleans 22 document.write(typeof true + "
"); // Prints: "boolean" 23 document.write(typeof false + "
"); // Prints: "boolean" 24 25 // Undefined 26 document.write(typeof undefined + "
"); // Prints: "undefined" 27 document.write(typeof undeclaredVariable + "
"); // Prints: "undefined" 28 29 // Null 30 document.write(typeof Null + "
"); // Prints: "object" 31 32 // Objects 33 document.write(typeof {name: "John", age: 18} + "
"); // Prints: "object" 34 35 // Arrays 36 document.write(typeof [1, 2, 4] + "
"); // Prints: "object" 37 38 // Functions 39 document.write(typeof function(){}); // Prints: "function" 40 </script> 41 </body> 42 </html> </pre>	<pre> number number number number number string string string boolean boolean undefined undefined undefined object object function </pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------

Figure 26 – The typeof Operator (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

As it is quite noticeable in the example from Figure 26, when the null value is tested using the typeof operator (line 22), "object" is returned instead of "null".

This is an enduring bug in JavaScript, but since lots of codes on the web are written around this behaviour, and since fixing it would generate many more problems, this issue was excluded by the committee that design and maintains JavaScript.

JavaScript Operators

Operators are symbols or keywords that inform the JavaScript engine to make a given action. For instance, the addition (+) symbol is an operator that tells JavaScript engine to add two variables or values, whereas the equal-to (==), greater-than (>) or less-than (<) symbols are the operators that tells JavaScript engine to compare two variables or values, etc.

Among the different operators used in JavaScript, the first ones to be described are the JavaScript **Arithmetic Operators**. These are put in action in order to perform common arithmetical operations (additions, subtraction, multiplication, and so on). The complete list follows below:

Operator	Description	Example	Result
+	Addition	$x + y$	Sum of x and y
-	Subtraction	$x - y$	Difference of x and y.
*	Multiplication	$x * y$	Product of x and y.
/	Division	x / y	Quotient of x and y
%	Modulus	$x \% y$	Remainder of x divided by y

Table 1 – Arithmetic Operators (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Here is a practical example on how the aforementioned operators can be used:


```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Arithmetic Operators</title>
6 </head>
7 <body>
8   <script>
9     var x = 10;
10    var y = 4;
11    document.write(x + y); // Prints: 14
12    document.write("<br>");
13
14    document.write(x - y); // Prints: 6
15    document.write("<br>");
16
17    document.write(x * y); // Prints: 40
18    document.write("<br>");
19
20    document.write(x / y); // Prints: 2.5
21    document.write("<br>");
22
23    document.write(x % y); // Prints: 2
24  </script>
25 </body>
26 </html>

```

Figure 27 – The JS Arithmetic Operators (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Then, there are as well **JS String Operators** (two of them):

Operator	Description	Example	Result
+	Concatenation	str1 + str2	Concatenation of str1 and str2
+=	Concatenation assignment	str1 += str2	Appends the str2 to the str1

Table 2 – String Operators (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

A practical example:


```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript String Operators</title>
6 </head>
7 <body>
8   <script>
9     var str1 = "Hello";
10    var str2 = " World!";
11
12    document.write(str1 + str2 + "<br>"); // Outputs: Hello World!
13
14    str1 += str2;
15    document.write(str1); // Outputs: Hello World!
16  </script>
17 </body>
18 </html>

```

Hello World!
Hello World!

Figure 28 – The JS Arithmetic Operators (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Concerning the **JS Incrementing and Decrementing Operators**, they are used to increment/decrement a variable's value.

Operator	Name	Effect
++x	Pre-increment	Increments x by one, then returns x
x++	Post-increment	Returns x, then increments x by one
--x	Pre-decrement	Decrements x by one, then returns x
x--	Post-decrement	Returns x, then decrements x by one

Table 3 – Incrementing and Decrementing Operators (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

A real-world example:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Incrementing and Decrementing Operators</title>
6 </head>
7 <body>
8   <script>
9     var x; // Declaring Variable
10
11     x = 10;
12     document.write(++x); // Prints: 11
13     document.write("<p>" + x + "</p>"); // Prints: 11
14
15     x = 10;
16     document.write(x++); // Prints: 10
17     document.write("<p>" + x + "</p>"); // Prints: 11
18
19     x = 10;
20     document.write(--x); // Prints: 9
21     document.write("<p>" + x + "</p>"); // Prints: 9
22
23     x = 10;
24     document.write(x--); // Prints: 10
25     document.write("<p>" + x + "</p>"); // Prints: 9
26   </script>
27 </body>
28 </html>

```

Figure 29 – JS Incrementing and Decrementing Operators (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

To combine conditional statements, **JS Logical Operators** are used.

Operator	Name	Example	Result
&&	And	x && y	True if both x and y are true
	Or	x y	True if either x or y is true
!	Not	!x	True if x is not true

Table 4 – Logical Operators (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Here is a practical example:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Logical Operators</title>
6 </head>
7 <body>
8   <script>
9     var year = 2018;
10
11     // Leap years are divisible by 400 or by 4 but not 100
12     if((year % 400 == 0) || ((year % 100 != 0) && (year % 4 == 0))){
13       document.write(year + " is a leap year.");
14     } else{
15       document.write(year + " is not a leap year.");
16     }
17   </script>
18 </body>
19 </html>

```

2018 is not a leap year.

Figure 30 – JS Incrementing and Decrementing Operators (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Regarding **JS Comparison Operators**, programmers use them to compare two values in a Boolean Fashion.

Operator	Name	Example	Result
==	Equal	x == y	True if x is equal to y
===	Identical	x === y	True if x is equal to y, and they are of the same <u>type</u>
!=	Not equal	x != y	True if x is not equal to y
!==	Not identical	x !== y	True if x is not equal to y, or they are not of the same type
<	Less than	x < y	True if x is less than y
>	Greater than	x > y	True if x is greater than y
>=	Greater than or equal to	x >= y	True if x is greater than or equal to y
<=	Less than or equal to	x <= y	True if x is less than or equal to y

Table 5 – Comparison Operators (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

JavaScript Events

Before going into deep on this section, it is important to acknowledge what an event is in this context. An event is something that occurs each time users interact with the web page, such as when a link or a button is clicked, text is entered into an input box or textarea, a selection is made in a select box, key is pressed on the keyboard, the mouse pointer is moved, a form is submitted, and so on. Every so often, the browser is able to trigger the events itself, for example when loading a page.

When an event happens, programmers can use a JavaScript event handler (or listener) to spot them and do specific task/s. By convention, the names for event handlers always start with the word "on", so an event handler for the click event is named onclick, likewise an event handler for the load event is named onload, event handler for the blur event is named onblur, etc.

There are many ways of assigning an event handler. The simplest way is to add them directly to the start tag of the HTML elements, by means of the special event-handler attributes. E.g., to assign a click handler for a button element, the onclick attribute may be used, as follows:

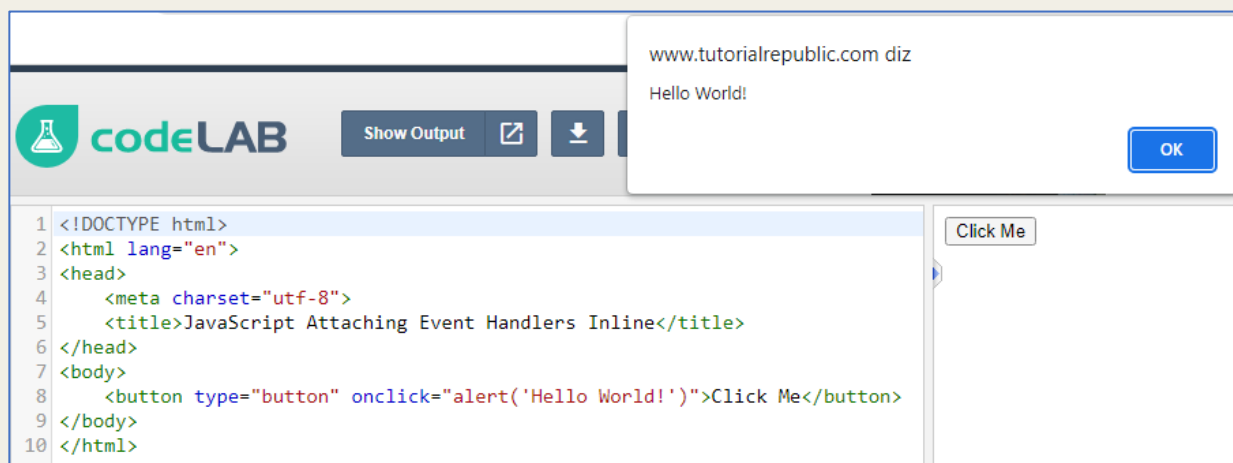


Figure 32 – JS Incrementing and Decrementing Operators (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Nonetheless, to keep the JavaScript detached from HTML, programmers can set up the event handler in an external JavaScript file or within the `<script>` and `</script>` tags, as follows:

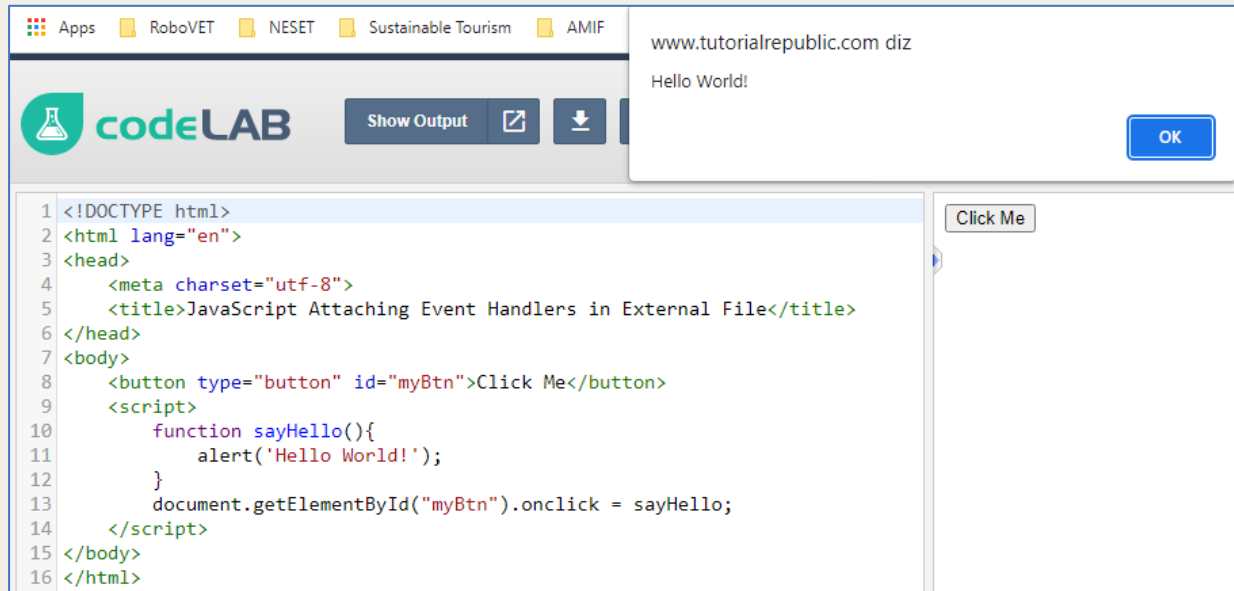


Figure 33 – Assigning an event handler (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Generally, events can be categorised into four main groups — **mouse events**, **keyboard events**, **form events** and **document/window events**. There are many other events, which will be exploited more furtherly. The most useful events will be overviewed below:

- **Mouse Events**

A mouse event is activated when the user clicks some element, moves the mouse pointer over an element, and so on. Some important mouse events and their event handlers are as follows:

- **The Click Event (onclick):** The click event happens when a user clicks on an element on a web page. Habitually, these are form elements and links. A click event can be handled with an onclick event handler.

The [following example](#) presents the case of an alert message popping up when the user clicks on the elements.

- **The Contextmenu Event (oncontextmenu):** it occurs when users click the mouse's right button on an element, opening a context menu. Oncontextmenu event handler handles a contextmenu event. [This example](#) pops an alert message when users right-click the elements.
- **The Mouseover Event (onmouseover):** it happens when users move the mouse pointer over an element. It can be handled with the onmouseover event handler. The [following example](#) shows an alert message when the mouse is placed over the elements.
- **The Mouseout Event (onmouseout):** it takes place when users move the mouse pointer outside of an element. It can be handled by using the onmouseout event handler. The [following example](#) will show you an alert message when the mouseout event occurs.

• Keyboard Events

A keyboard event takes place when the user presses or releases a key on the keyboard. Some of the most important keyboard events and their event handlers are as follows:

- **The Keydown Event (onkeydown):** it happens when users press down a key on the keyboard. It can be handled by using the onkeydown event handler. [This example](#) shows an alert message each time a keydown event occurs.
- **The Keyup Event (onkeyup):** it occurs when users release a key on the keyboard. Handled with onkeyup event handler. [This example](#) shows an alert message when it occurs.
- **The Keypress Event (onkeypress):** it happens when a user presses down a key on the keyboard that has a character value linked to it. E.g., Ctrl, Shift, Alt, Esc, Arrow keys, etc. will not generate a keypress event, but will generate a keydown and keyup event. The onkeypress event

handler handles the keypress event. It can be checked [here](#) through an alert message.

• Form Events

A form event is triggered as soon as a form control receives or loses focus or when the user modifies a form control value (e.g., by typing text in a text input), select an option in a select box, etc.

- **The Focus Event (onfocus):** it happens when users give focus to an element on a webpage. It can be handled with onfocus event handler. [This example](#) will highlight the background of text input in yellow colour when it obtains the focus.
- **The Blur Event (onblur):** it happens when the user takes the focus away from a window or form element. It can be handled with the onblur event handler. The following [example](#) will exhibit an alert message as soon as the text input element misses focus.
- **The Change Event (onchange):** it occurs as soon as a user modifies the value of a form element. Event handler: onchange. This [example](#) shows an alert message when the option in the select box is changed.
- **The Submit Event (onsubmit):** it only happens when the user submits a form on a webpage. Event handler: onsubmit. This [example](#) shows an alert message while submitting the form.

• Document/Window Events

Situations in which the page has loaded or resized the browser window can as well trigger events.

- **The Load Event (onload):** it happens when a webpage fully loads in a web browser. Event handler: onload. The [following](#) example will emit an alert message as soon as page loads.

- **The Unload Event (onunload):** when the user leaves the present webpage, this event takes place. Event handler: onunload. [This example](#) shows an alert pop-up when the user tries to leave the page.
- **The Resize Event (onresize):** it happens when the Internet user resizes, minimises or maximises the browser window. Event handler: onresize. [This example](#) shows an alert message right after the user resizes the browser's window.

JavaScript Strings

In JavaScript, strings play a key role on the overall structure of a webpage, since they are a sequence of letters, numbers, special characters and arithmetic values or even a combination of all. They can be created by enfolding the string literal (i.e. string characters) either within single quotes (') or double quotes ("), as follows:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>Creating Strings in JavaScript</title>
6 </head>
7 <body>
8   <script>
9     // Creating variables
10    var myString = 'Hello World!'; // Single quoted string
11    var myString = "Hello World!"; // Double quoted string
12
13    // Printing variable values
14    document.write(myString + "<br>");
15    document.write(myString);
16  </script>
17 </body>
18 </html>
```

Hello World!
Hello World!

Figure 34 – JavaScript strings example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Quotes can be used inside a string, but they should not match the quotes surrounding the string:

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>Using Quotes inside JavaScript Strings</title> 6 </head> 7 <body> 8 <script> 9 // Creating variables 10 var str1 = "it's okay"; 11 var str2 = 'He said "Goodbye"'; 12 var str3 = "She replied 'Calm down, please'"; 13 14 // Printing variable values 15 document.write(str1 + "
"); 16 document.write(str2 + "
"); 17 document.write(str3); 18 </script> 19 </body> 20 </html> </pre>	<pre> it's okay He said "Goodbye" She replied 'Calm down, please' </pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------

Figure 35 – JavaScript strings example II (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Nonetheless, single quotes can still be put inside a single quoted strings or double quotes inside double quoted strings, by separating the quotes with a backslash character (\), as *Figure 36* shows. The backslash is termed an **escape character**, and the sequences \ ' and \ " are **escape sequences**.

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>Escaping Quotes inside JavaScript Strings</title> 6 </head> 7 <body> 8 <script> 9 // Creating variables 10 var str1 = 'it\'s okay'; 11 var str2 = "He said \"Goodbye\""; 12 var str3 = 'She replied \'Calm down, please\''; 13 14 // Printing variable values 15 document.write(str1 + "
"); 16 document.write(str2 + "
"); 17 document.write(str3); 18 </script> 19 </body> 20 </html> </pre>	<pre> it's okay He said "Goodbye" She replied 'Calm down, please' </pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------

Figure 36 – The backslash (\) escape character (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Escape sequences are likewise valuable for adding characters that cannot be inserted by means of a keyboard. Some other most frequently used escape sequences are:

- `\n` is replaced by the newline character
- `\t` is replaced by the tab character
- `\r` is replaced by the carriage-return character
- `\b` is replaced by the backspace character
- `\\` is replaced by a single backslash (`\`)

Figure 37 elucidates how escape sequences work:

<pre><!DOCTYPE html> <html lang="en"> <head> <meta charset="utf-8"> <title>JavaScript Escape Sequences</title> </head> <body> <script> // Creating variables var str1 = "The quick brown fox \n jumps over the lazy dog."; document.write("<pre>" + str1 + "</pre>"); // Create line break var str2 = "C:\\Users\\Downloads"; document.write(str2 + "
"); // Prints C:UsersDownloads var str3 = "C:\\\\Users\\\\Downloads"; document.write(str3); // Prints C:\\Users\\Downloads </script> </body> </html></pre>	<pre>The quick brown fox jumps over the lazy dog. C:UsersDownloads C:\\Users\\Downloads</pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------

Figure 37 – How escape sequences work (Source: <https://www.tutorialpublic.com/javascript-tutorial/javascript-get-started.php>)

Performing Operations on Strings

JavaScript makes available several properties and methods to make operations on string values. Precisely, only objects can have properties and methods. However, in JavaScript, primitive data types can perform like objects when the programmer refers to them with the property access notation. JavaScript offers this possibility through the creation of a provisional wrapper object for primitive data types. This procedure is done automatically by the JS interpreter in the background.

Getting the Length of a String

The length property returns the length of the string, which is the number of characters delimited in the string, including the number of special characters as well, such as `\t` or `\n`. Programmers should be careful on using parentheses after length (e.g. `str.length()`), as the correct way is `str.length` (or else, it will generate an error).

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>Get String Length in JavaScript</title>
6 </head>
7 <body>
8   <script>
9     var str1 = "This is a paragraph of text.";
10    document.write(str1.length + "<br>"); // Prints 28
11
12    var str2 = "This is a \n paragraph of text.";
13    document.write(str2.length); // Prints 30, because \n is only one
    character
14  </script>
15 </body>
16 </html>
    
```

Figure 38 – How to get the length of a string (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Finding a String Inside Another String

`indexOf()` method can be used to find a substring or string within another string. This technique returns the index or position of the first incidence of a specified string within a string.

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Find the Position of Substring within a String</title>
6 </head>
7 <body>
8   <script>
9     var str = "If the facts don't fit the theory, change the facts.";
10    var pos = str.indexOf("facts");
11    document.write(pos); // Outputs: 7
12  </script>
13 </body>
14 </html>
    
```

Figure 39 – Finding a string inside another string (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Likewise, the `lastIndexOf()` technique can be used to get the index or position of the last occurrence of the specified string within a string, as follows:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Find the Position of Substring within a String</title>
6 </head>
7 <body>
8   <script>
9     var str = "If the facts don't fit the theory, change the facts.";
10    var pos = str.lastIndexOf("facts");
11    document.write(pos); // Outputs: 46
12  </script>
13 </body>
14 </html>
```

Figure 40 – Finding a string inside another string using the `lastIndexOf()` method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Both the `indexOf()`, and the `lastIndexOf()` approaches return -1 if the substring is not found. Both methods as well accept an optional integer parameter which stipulates the position within the string at which to start the search.

Searching for a Pattern Inside a String

The `search()` method can be used to search a particular piece of text or pattern inside a string. As the `indexOf()` approach, `search()` also returns the index of the first match, and returns -1 if no matches were found, but unlike `indexOf()`, `search()` can as well take a *regular expression* as its argument to deliver advanced search aptitudes. It should be stated that the `search()` approach does not support global searches, as it disregards the `g` flag or modifier of its regular expression argument.

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Search Text or Pattern inside a String</title>
6 </head>
7 <body>
8   <script>
9     var str = "Color red looks brighter than color blue.";
10
11     // Case sensitive search
12     var pos1 = str.search("color");
13     document.write(pos1 + "<br>"); // Outputs: 30
14
15     // Case insensitive search using regexp
16     var pos2 = str.search(/color/i);
17     document.write(pos2); // Outputs: 0
18   </script>
19 </body>
20 </html>
    
```

Figure 41 – Searching for a pattern inside a string (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Extracting a Substring from a String

For take out a part or substring from a string, the slice() method can be used. This takes two parameters: *start index* (index where extraction begins), and an optional *end index* (index before which to end extraction), like str.slice(startIndex, endIndex).

The example below slices out a portion of a string from position 4 to position 15:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Slice Out a Portion of a String</title>
6 </head>
7 <body>
8   <script>
9     var str = "The quick brown fox jumps over the lazy dog.";
10    var subStr = str.slice(4, 15);
11    document.write(subStr); // Prints: quick brown
12  </script>
13 </body>
14 </html>
    
```

Figure 42 – Extracting a Substring from a String using the slice() method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Negative values can be specified as well. These values are treated as `strLength + startIndex`, where `strLength` is the length of the string (for instance, `str.length`), for example, if `startIndex` is `-5` it is treated as `strLength - 5`. If `startIndex` is greater than or equal to the length of the string, `slice()` method returns an empty string. Correspondingly, if optional `endIndex` is not specified or omitted, the `slice()` method extracts to the end of the string.

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Slice Strings Using Negative Indexes</title>
6 </head>
7 <body>
8   <script>
9     var str = "The quick brown fox jumps over the lazy dog.";
10    document.write(str.slice(-28, -19) + "<br>"); // Prints: fox jumps
11    document.write(str.slice(31)); // Prints: the lazy dog.
12  </script>
13 </body>
14 </html>
```




Figure 43 – Specifying negative values (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

The `substring()` method to extract a section of the given string based on start and end indexes, as `str.substring(startIndex, endIndex)`. The `substring()` method is very comparable to the `slice()`, except some differences:

- If either argument is less than 0 or is NaN, it is treated as 0.
- If either argument is greater than `str.length`, it is treated as if it were `str.length`.
- If `startIndex` is greater than `endIndex`, then `substring()` will switch those two arguments; i.e., `str.substring(5, 0) == str.substring(0, 5)`.

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Extract substring from a String</title> 6 </head> 7 <body> 8 <script> 9 var str = "The quick brown fox jumps over the lazy dog."; 10 document.write(str.substring(4, 15) + "
"); // Prints: quick brown 11 document.write(str.substring(9, 0) + "
"); // Prints: The quick 12 document.write(str.substring(-28, -19) + "
"); // Prints nothing 13 document.write(str.substring(31)); // Prints: the lazy dog. 14 </script> 15 </body> 16 </html> </pre>	quick brown The quick the lazy dog.
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------

Figure 44 – The `substring()` method to extract a section of the given string based on start and end indexes
 (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Extracting a Fixed Number of Characters from a String

JavaScript also delivers the `substr()` technique, which is similar to `slice()` with a minor difference: the second parameter stipulates the number of characters to extract instead of ending index, as `str.substr(startIndex, length)`. If length is 0 or a negative number, an empty string is returned.

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Extract Fixed Number of Characters from a String</title> 6 </head> 7 <body> 8 <script> 9 var str = "The quick brown fox jumps over the lazy dog."; 10 document.write(str.substr(4, 15) + "
"); // Prints: quick brown fox 11 document.write(str.substr(-28, -19) + "
"); // Prints nothing 12 document.write(str.substr(-28, 9) + "
"); // Prints: fox jumps 13 document.write(str.substr(31)); // Prints: the lazy dog. 14 </script> 15 </body> 16 </html> </pre>	quick brown fox fox jumps the lazy dog.
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------

Figure 45 – Extracting a Fixed Number of Characters from a String (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Replacing the Contents of a String

The `replace()` technique is used to replace part of a string with another string. This approach takes a regular expression to match or substring with two parameters and a replacement string, i.e. `str.replace(regex|substr, newSubstr)`. This `replace()` method returns a new string, it does not distress the original string, which will remain unaffected.

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Replace Part of a String with another String</title> 6 </head> 7 <body> 8 <script> 9 var str = "Color red looks brighter than color blue."; 10 var result = str.replace("color", "paint"); 11 document.write(result); // Outputs: Color red looks brighter than paint blue. 12 </script> 13 </body> 14 </html> </pre>	<p>Color red looks brighter than paint blue.</p>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------

Figure 46 – Replacing the Contents of a String using the `replace()` technique (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

By default, the `replace()` technique substitutes only the first match, and it is case-sensitive. To replace the substring within a string in a case-insensitive way, a *regular expression (regex)* with an `i` modifier can be used, as follows:

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Replace Part of a String with another String</title> 6 </head> 7 <body> 8 <script> 9 var str = "Color red looks brighter than color blue."; 10 var result = str.replace(/color/i, "paint"); 11 document.write(result); // Outputs: paint red looks brighter than color blue. 12 </script> 13 </body> 14 </html> </pre>	<p>paint red looks brighter than color blue.</p>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------

Figure 46 – Replacing the Contents of a String using the `replace()` technique (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Also, to replace all the incidences of a substring within a string in a case-insensitive manner, the g modifier along with the i modifier can be used, as follows below:

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Replace All Occurrences of a Substring in a String</title> 6 </head> 7 <body> 8 <script> 9 var str = "Color red looks brighter than color blue."; 10 var result = str.replace(/color/ig, "paint"); 11 document.write(result); // Outputs: paint red looks brighter than paint blue. 12 </script> 13 </body> 14 </html> </pre>	<p>paint red looks brighter than paint blue.</p>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------

Figure 47 – Replacing all the incidences of a substring within a string in a case-insensitive manner (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Converting a String to Uppercase or Lowercase

The toUpperCase() method is used to convert a string to uppercase, as follows:

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Convert a String to Uppercase Characters</title> 6 </head> 7 <body> 8 <script> 9 var str = "Hello World!"; 10 var result = str.toUpperCase(); 11 document.write(result); // Prints: HELLO WORLD! 12 </script> 13 </body> 14 </html> </pre>	<p>HELLO WORLD!</p>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------

Figure 48 – Converting a String to Uppercase using toUpperCase() method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Likewise, the toLowerCase() method is used to convert a string to lowercase:

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Convert a String to Lowercase Characters</title> 6 </head> 7 <body> 8 <script> 9 var str = "Hello World!"; 10 var result = str.toLowerCase(); 11 document.write(result); // Prints: hello world! 12 </script> 13 </body> 14 </html> </pre>	<p>hello world!</p>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------

Figure 49 – Converting a String to Lowercase using `toLowerCase()` method (**Source:** <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Concatenating Two or More Strings

Two or more strings can be concatenated or combined by using the `+` and `+=` assignment operators.

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Join Two or More Strings</title> 6 </head> 7 <body> 8 <script> 9 var hello = "Hello"; 10 var world = "World"; 11 var greet = hello + " " + world; 12 document.write(greet + "
"); // Prints: Hello World 13 14 var wish = "Happy"; 15 wish += " New Year"; 16 document.write(wish); // Prints: Happy New Year 17 </script> 18 </body> 19 </html> </pre>	<p>Hello World Happy New Year</p>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------

Figure 50 – Concatenating Two or More Strings (**Source:** <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Accessing Individual Characters from a String

The `charAt()` method can be used to access individual character from a string, as `str.charAt(index)`. The index specified should be a number between 0 and `str.length - 1`. If no index is given, the first character in the string is returned, since the default is 0.

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Extract a Single Character from a String</title>
6 </head>
7 <body>
8   <script>
9     var str = "Hello World!";
10    document.write(str.charAt() + "<br>"); // Prints: H
11    document.write(str.charAt(6) + "<br>"); // Prints: W
12    document.write(str.charAt(30) + "<br>"); // Prints nothing
13    document.write(str.charAt(str.length - 1)); // Prints: !
14  </script>
15 </body>
16 </html>

```

Figure 51 – Accessing Individual Characters from a String (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

However, there is a good alternative to this procedure. Since ECMAScript 5, strings can be treated like read-only arrays, and individual characters can be seen from a string using square brackets (`[]`) instead of the `charAt()` approach, as follows below:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Extract a Single Character from a String</title>
6 </head>
7 <body>
8   <script>
9     var str = "Hello World!";
10    document.write(str[0] + "<br>"); // Prints: H
11    document.write(str[6] + "<br>"); // Prints: W
12    document.write(str[str.length - 1] + "<br>"); // Prints: !
13    document.write(str[30]); // Prints: undefined
14  </script>
15 </body>
16 </html>

```

Figure 52 – Accessing Individual Characters from a String using square brackets (`[]`) instead of the `charAt()` approach (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

Splitting a String into an Array

The `split()` method can be used to fragment a string into an array of strings, using the syntax `str.split(separator, limit)`. The separator argument stipulates the string at which each split should happen, while the limit arguments specifies the maximum length of the array. If separator argument is omitted or not found in the specified string, the entire string is allocated to the first element of the array.

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Split a String into an Array</title>
6 </head>
7 <body>
8   <script>
9     var fruitsStr = "Apple, Banana, Mango, Orange, Papaya";
10    var fruitsArr = fruitsStr.split(", ");
11    document.write(fruitsArr[0] + "<br>"); // Prints: Apple
12    document.write(fruitsArr[2] + "<br>"); // Prints: Mango
13    document.write(fruitsArr[fruitsArr.length - 1]); // Prints: Papaya
14    document.write("<hr>");
15
16    // Loop through all the elements of the fruits array
17    for(var i in fruitsArr) {
18      document.write("<p>" + fruitsArr[i] + "</p>");
19    }
20  </script>
21 </body>
22 </html>
```

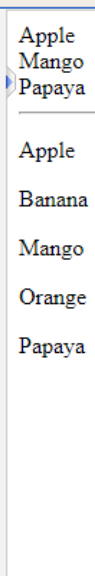


Figure 53 – Splitting a String into an Array (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-get-started.php>)

To split a string into an array of characters, an empty string ("") should be stipulated as a separator, as [this example](#) shows.

JavaScript Numbers

There are two types of numbers in JavaScript:

- **Regular numbers** in JavaScript are stored in 64-bit format IEEE-754, otherwise known as “*double precision floating point numbers*”. These are the most commonly used type of numbers.
- **BigInt numbers**, to represent integers of random length. They are sometimes needed, because a regular number cannot securely exceed 2^{53} or be less than -2^{53} .

There are more ways to write a number. For instance, the obvious way to write 1 billion would be ‘1000000000’ or ‘1_000_000_000’, using the underscore as a separator. In this case, the underscore is the “syntactic sugar”, meaning that it makes the number clearer to read. The JS engine ignores underscores between digits, so it is the exact same one billion from the first case.

However, in the real world, everyone most certainly will try to avoid writing long sequences of zeros. Something like “1bn” for a billion or “2.5 bn” for 2 billion 500 million seems more reasonable. The same principle applies for most large numbers. That being said, it is possible to shorten a number in JS, by adding the letter “e” to it and specifying the quantity of zeroes:

```
1 let billion = 1e9; // 1 billion, literally: 1 and 9 zeroes
2
3 alert( 7.3e9 ); // 7.3 billions (same as 7300000000 or 7_300_000_000)
```

Figure 54 – Specifying the quantity of zeroes (Source: <https://javascript.info/number>)

So, e multiplies the number by 1 with the given zeroes count.

```
1e3 === 1 * 1000; // e3 means *1000
1.23e6 === 1.23 * 1000000; // e6 means *1000000
```

The same principle applies for small numbers. For instance, what should be written for 1 microsecond (one millionth of a second)?

```
let mcs = 0.000001;
```

Just like the aforementioned case with the big numbers, using "e" can be helpful. To avoid writing the zeroes explicitly, the following can be written:

```
let mcs = 1e-6; // six zeroes to the left from 1
```

There are 6 zeroes in 0.000001. Then, it is not hard to conclude 1e-6.

Therefore, a negative number after "e" implies a division by 1 with the given number of zeroes, as follows:

```
1 // -3 divides by 1 with 3 zeroes
2 1e-3 === 1 / 1000; // 0.001
3
4 // -6 divides by 1 with 6 zeroes
5 1.23e-6 === 1.23 / 1000000; // 0.00000123
```

Figure 55 – The division of 1 by the given number of zeroes (Source: <https://javascript.info/number>)

Hex, binary, and octal numbers

Hexadecimal numbers are commonly utilized in JavaScript to embody colours, encode characters, and for many other purposes. So obviously, there is a quicker way to write them: 0x and then the number, as follows:

```
1 alert( 0xff ); // 255
2 alert( 0xFF ); // 255 (the same, case doesn't matter)
```

Figure 56 – The 0x way to speed up JS processes (Source: <https://javascript.info/number>)

Binary and octal numeral systems are not often utilized, but as well supported using the 0b and 0o (zero, o) prefixes:

```
1 let a = 0b11111111; // binary form of 255
2 let b = 0o377; // octal form of 255
3
4 alert( a == b ); // true, the same number 255 at both sides
```

Figure 57 – The 0b and 0o prefixes for binary and octal numeral systems (Source: <https://javascript.info/number>)

There are merely 3 numeral systems with this support. For further numeral systems, the function `parseInt` should be used.

toString(base) method

The `num.toString(base)` method returns a string representation of `num` in the numeral system with the provided base, as follows:

```
1 let num = 255;
2
3 alert( num.toString(16) ); // ff
4 alert( num.toString(2) ); // 11111111
```

Figure 58 – The `num.toString(base)` method (Source: <https://javascript.info/number>)

The base can vary from 2 to 36. By default, it's 10.

There are general use cases for this:

- **base=16** is applied for hex colours, character encodings etc, digits can be 0..9 or A..F.
- **base=2** is mainly for correcting bitwise operations, digits can be 0 or 1.
- **base=36** is the maximum, digits can be 0..9 or A..Z. The entire Latin alphabet is used to represent a number. A curious, but useful case for 36 is when it is needed to turn a long numeric identifier into something shorter, for example to make a short url.

There is a specific case to be noted. When two dots are spotted in `123456..toString(36)`, this is not a typo. When the programmer wants to call a method directly on a number, it is needed to place two dots (`..`) after it.

If a single dot is placed as follows (`'123456.toString(36)'`), it would be an error, as JavaScript syntax entails the decimal part after the first dot. And if the programmer places one more dot, then JavaScript knows that the decimal part is empty and now goes the method.

Rounding

One of the most applied operations when operating with numbers is **rounding**.

There are some built-in functions for rounding:

- **Math.floor**

Rounds down: 3.1 turn into 3, and -1.1 converts -2.

- **Math.ceil**

Rounds up: 3.1 becomes 4, and -1.1 converts -1.

- **Math.round**

Rounds to the nearest integer: 3.1 becomes 3, 3.6 becomes 4, the middle case: 3.5 rounds up to 4 too.

- **Math.trunc (not supported by Internet Explorer)**

Eliminates anything after the decimal point without rounding: 3.1 turn out to be 3, -1.1 becomes -1.

These functions cover all the potential ways to cope with the decimal part of a number.

But what about rounding the number to n-th digit after the decimal?

For example, to round 1.2345 to two digits (1.23).

There are two ways to do so:

1. **Multiply-and-divide.**

E.g., to round the number to the 2nd digit after the decimal, one can multiply the number by 100 (or a bigger power of 10), call up the rounding function and then divide it back.

- The **toFixed(n)** method rounds the number to n digits after the point and returns a string representation of the result.

```
1 let num = 12.34;  
2 alert( num.toFixed(1) ); // "12.3"
```

Figure 58 – The toFixed(n) method (Source: <https://javascript.info/number>)

This rounds up or down to the nearest value, like Math.round:

```
1 let num = 12.36;  
2 alert( num.toFixed(1) ); // "12.4"
```

Figure 59 – The toFixed(n) method – continuation (Source: <https://javascript.info/number>)

It should be stated that result of toFixed is a string. If the decimal part is shorter than required, zeroes are attached to the end:

```
1 let num = 12.34;  
2 alert( num.toFixed(5) ); // "12.34000", added zeroes to make exactly 5 digits
```

Figure 60 – The toFixed(n) method – final exposition (Source: <https://javascript.info/number>)

Imprecise calculations

On the inside, a number is represented in 64-bit format **IEEE-754**, so there are precisely 64 bits to store a number: 52 to store the digits; 11 to store the position of the decimal point (zero for integer numbers), and 1 bit is for the sign.

If a number is too big, it would pour out the 64-bit storage, possibly giving an infinity.

Something that ends to happen quite often is the loss of precision.

Considering the following test:

```
1 alert( 0.1 + 0.2 == 0.3 ); // false
```

Figure 61 – First test for imprecise calculations (Source: <https://javascript.info/number>)

As it looks, strangely the result from the $0.1+0.2$ sum is a false 0.3. What could it be, then?

```
1 alert( 0.1 + 0.2 ); // 0.30000000000000004
```

Figure 62 – Second test for imprecise calculations (Source: <https://javascript.info/number>)

As it could be imagined, there are more consequences than an incorrect comparison in this case. Pretending that a purchase is being made on an online shop and the visitor puts \$0.10 and \$0.20 goods into his/her cart... the total amount of the order would be \$0.30000000000000004. That would be quite surprising.

Why does a situation like this happen?

A number is stored in memory in its binary form, a sequence of bits – ones and zeroes. However, fractions like 0.1, 0.2 that seem simple in the decimal numeric system are in fact unending fractions in their binary form.

That is to say, what is 0.1? It is one divided by ten $1/10$, one-tenth. In decimal numeral system such numbers are easily representable. Comparing it to one-third: $1/3$. It becomes an endless fraction 0.33333(3).

So, division by powers 10 is guaranteed to work well in the decimal system, but division by 3 is not. For the same reason, in the binary numeral system, the division by powers of 2 is guaranteed to work, but $1/10$ becomes an endless binary fraction.

There is just no way to store exactly 0.1 or exactly 0.2 using the binary system, just like there is no way to store one-third as a decimal fraction.

The numeric format IEEE-754 resolves this by rounding to the closest possible number. These rounding rules normally do not allow one to see that “tiny precision loss”, but it exists.

```
1 alert( 0.1.toFixed(20) ); // 0.10000000000000000555
```

Figure 63 – Final test for imprecise calculations (Source: <https://javascript.info/number>)

When two numbers are summed, their “precision losses” come together. Therefore 0.1 + 0.2 is not 0.3 precisely.

The most reliable method to cope with this situation is **toFixed(n)**:

```
1 let sum = 0.1 + 0.2;  
2 alert( sum.toFixed(2) ); // 0.30
```

Figure 64 – The toFixed(in) method (Source: <https://javascript.info/number>)

It should be stated that toFixed(n) returns a string every time. It ensures that it has 2 digits after the decimal point. That is quite useful if there is need to show \$0.30 for e-shopping purposes.

Tests: isFinite and isNaN

Previously, those two special numeric values have been described. **Infinity** (and **-Infinity**) is a special numeric value that is greater (less) than anything. **NaN** represents an error. Both belong to the type **number**, but are not “normal” numbers, so there are special functions to check for them:

- **isNaN(value)** converts its argument to a number and then tests it for being **NaN**:

```
1 alert( isNaN(NaN) ); // true  
2 alert( isNaN("str") ); // true
```

Figure 64 – The isNaN(value) method (Source: <https://javascript.info/number>)

- **isFinite(value)** converts its argument to a number and returns **true** if it's a regular number, not **NaN/Infinity/-Infinity**;

```
1 alert( isFinite("15") ); // true
2 alert( isFinite("str") ); // false, because a special value: NaN
3 alert( isFinite(Infinity) ); // false, because a special value: Infinity
```

Figure 65 – The `isFinite(value)` method (Source: <https://javascript.info/number>)

parseInt and parseFloat

Numeric conversion using a plus **+** or **Number()** is rigorous. If a value is not exactly a number, it fails. The only exception is spaces at the beginning or at the end of the string, as they are ignored.

However, in real life there are often values in units, as "100px" or "12pt" in CSS. Likewise, in many countries the currency symbol goes after the amount, so having the example of "19€", the programmer would like to extract a numeric value out of that.

That's what **parseInt** and **parseFloat** are for.

They "read" a number from a string until they cannot. In case of an error, the collected number is returned. The function **parseInt** returns an integer, whereas **parseFloat** will return a floating-point number:

```
1 alert( parseInt('100px') ); // 100
2 alert( parseFloat('12.5em') ); // 12.5
3
4 alert( parseInt('12.3') ); // 12, only the integer part is returned
5 alert( parseFloat('12.3.4') ); // 12.3, the second point stops the reading
```

Figure 66 – `parseInt` and `parseFloat` (Source: <https://javascript.info/number>)

Other math functions

JavaScript owns a built-in Math object which includes a small library of mathematical functions and constants:

- **Math.random()**

Returns a random number from 0 to 1 (not including 1).

- **Math.max(a, b, c...) / Math.min(a, b, c...)**

Returns the greatest/smallest from the arbitrary number of arguments.

- **Math.pow(n, power)**

Returns n raised to the given power.

JavaScript If...Else Statements

As many other programming languages, JavaScript also allows to write code that execute different actions based on the results of a logical or comparative test conditions at run time. Thus, test conditions can be created as expressions that evaluates to either true or false and, based on these results, certain actions can be performed.

There are several conditional statements in JavaScript that can be used to make decisions:

- The **if** statement;
- The **if...else** statement;
- The **if...else if....else** statement;
- The **switch...case** statement.

The if statement

The if statement is used to execute a block of code only if the specified condition is evaluated as being true. This is the simplest JS's conditional statements and can be written as:


```
if(condition) {  
    // Code to be executed  
}
```

Figure 67 – Code to create an if condition (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-if-else-statements.php>)

A practical example can be quite demonstrative of this feature’s usefulness. If the current day is Friday, the coding shown on *Figure 68* will display a message saying “Have a nice weekend”:

```
1 <!DOCTYPE html>  
2 <html lang="en">  
3 <head>  
4   <meta charset="utf-8">  
5   <title>JavaScript IF Statement</title>  
6 </head>  
7 <body>  
8   <script>  
9     var now = new Date();  
10    var dayOfWeek = now.getDay(); // Sunday - Saturday : 0 - 6  
11  
12    if(dayOfWeek == 5) {  
13      document.write("Have a nice weekend!");  
14    }  
15  </script>  
16  <p><strong>Note:</strong> This example will print "Have a nice weekend!" if the  
17  current day is Friday.</p>  
18 </body>  
19 </html>
```

Note: This example will print "Have a nice weekend!" if the current day is Friday.

Figure 68 – If...statement created to display a “Have a nice weekend” message in case is Friday (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-if-else-statements.php>)

The if...else statement

The decision-making process of JS can be enhanced by providing an alternative choice through adding an **else statement** to the **if statement**. The if...else statement allows to execute one block of code if the specified condition evaluates as *true* and another block of code if it evaluates as *false*.

```
if(condition) {  
    // Code to be executed if condition is true  
} else {  
    // Code to be executed if condition is false  
}
```

Figure 69 – Code to create an if...else condition (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-if-else-statements.php>)

The example from *Figure 68* will be applied to test the `if...else` statement. This time, the “Have a nice weekend!” message will be displayed in case the current day is Friday (as the previous case), however, it will emit the “Have a nice day” message if it is not Friday.

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript If-Else Statement</title>
6 </head>
7 <body>
8   <script>
9     var now = new Date();
10    var dayOfWeek = now.getDay(); // Sunday - Saturday : 0 - 6
11
12    if(dayOfWeek == 5) {
13      document.write("Have a nice weekend!");
14    } else {
15      document.write("Have a nice day!");
16    }
17  </script>
18 </body>
19 </html>
```

Have a nice day!

Figure 70 – Code to create an `if...else` condition, displaying a “Have a nice day” or “Have a nice weekend” message depending on the case (**Source:** <https://www.tutorialrepublic.com/javascript-tutorial/javascript-if-else-statements.php>)

The Ternary Operator

The ternary operator gives a **shorthand way of writing the `if...else` statements**. It is characterized by the question mark (“?”) symbol and it takes three operands: a condition to check, a result for true, and a result for false. Its basic syntax is as follows:

```
var result = (condition) ? value1 : value2
```

If the condition is evaluated as *true*, the `value1` will be returned, if not `value2` will be returned.

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Typical Conditional Statement</title>
6 </head>
7 <body>
8   <script>
9     var userType;
10    var age = 21;
11    if(age < 18) {
12      userType = 'Child';
13    } else {
14      userType = 'Adult';
15    }
16    document.write(userType); // Prints Adult
17  </script>
18 </body>
19 </html>

```

Figure 71 – How to apply the ternary operator (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-if-else-statements.php>)

JavaScript Switch...Case Statements

The **switch..case** statement is an alternate scenario to the **if...else if...else** statement, which does nearly the same thing. The **switch...case** statement analyses a variable or expression against a series of values until it finds a match, and then executes the block of code corresponding to that match. Its syntax is as follows:

```

switch(x){
  case value1:
    // Code to be executed if x === value1
    break;
  case value2:
    // Code to be executed if x === value2
    break;
  ...
  default:
    // Code to be executed if x is different from all values
}

```

Figure 72 – Syntax for switch...case statements (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-if-else-statements.php>)

[This example](#) displays the name of the day of the week the reader is in.

The **switch...case** statement diverges from the **if...else** statement in one crucial way. The switch statement executes line by line and when JavaScript finds a case clause that evaluates to *true*, it does not only performs the code corresponding to that case clause, but also executes all the successive case clauses till the end of the switch block automatically.

To prevent this, a break statement must be included after each case (as can be checked on *Figure 73*). The break statement informs the JS interpreter to break out of the **switch...case** statement block once it executes the code linked to the first *true* case.

The break statement is yet not required for the case or default clause when it appears at last in a switch statement. However, it is a good programming practice to dismiss the last case, or default clause in a switch statement with a break. It prevents a possible programming error later on if another case statement is included in the switch statement.

The default clause is voluntary, which stipulates the actions to be made if no case matches the switch expression. The default clause does not have to be the last clause to be found in a switch statement.

Each case value must be exclusive within a switch statement. Still, different cases do not need to have a unique action. Several cases can share the same action.

JavaScript Arrays

Arrays are complex variables that allow to store more than one value or a group of values under a single variable name. JavaScript arrays can store any valid value, including strings, numbers, objects, functions, and at the same time other arrays, therefore enabling to create more complex data structures such as an array of objects or an array of arrays.

In the following example, the name of colours will be stored in a JavaScript code:

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Storing Single Values</title> 6 </head> 7 <body> 8 <script> 9 // Creating variables 10 var color1 = "Red"; 11 var color2 = "Green"; 12 var color3 = "Blue"; 13 14 // Printing variable values 15 document.write(color1 + "
"); 16 document.write(color2 + "
"); 17 document.write(color3); 18 </script> 19 </body> 20 </html> </pre>	<p>Red Green Blue</p>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------

Figure 73– Storing the name of colours in a JS code (**Source:** <https://www.tutorialrepublic.com/javascript-tutorial/javascript-arrays.php>)

However, it may be quite difficult and tiring to store multiple elements in variables, opposite to only three in the case above. Additionally, using so many variables at the same time and keeping track of them all will be a very tricky task. And here array comes into play. Arrays solve this problem by providing an ordered structure for storing multiple values or a group of values.

Creating an Array

The easiest way to create an array in JavaScript is enclosing a comma-separated list of values in square brackets ([]), as shown in the following syntax:

```
var myArray = [element0, element1, ..., elementN];
```


<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>Creating Arrays in JavaScript</title> 6 </head> 7 <body> 8 <script> 9 // Creating variables 10 var colors = ["Red", "Green", "Blue"]; 11 var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"]; 12 var cities = ["London", "Paris", "New York"]; 13 var person = ["John", "Wick", 32]; 14 15 // Printing variable values 16 document.write(colors + "
"); 17 document.write(fruits + "
"); 18 document.write(cities + "
"); 19 document.write(person); 20 </script> 21 </body> 22 </html> </pre>	<pre> Red,Green,Blue Apple,Banana,Mango,Orange,Papaya London,Paris,New York John,Wick,32 </pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------

Figure 74 – Creating an Array (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-arrays.php>)

Accessing the elements of an Array

Array elements can be accessed by their index utilizing the square bracket notation. An index is a number that represents an element's position in an array.

Array indexes are zero-based. This means that the first item of an array is stored at index 0, not 1, the second item is stored at index 1, and so on. Array indexes start at 0 and go up to the number of elements minus 1. So, array of five elements would have indexes from 0 to 4.

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Access Individual Elements of an Array</title> 6 </head> 7 <body> 8 <script> 9 var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"]; 10 11 document.write(fruits[0] + "
"); // Prints: Apple 12 document.write(fruits[1] + "
"); // Prints: Banana 13 document.write(fruits[2] + "
"); // Prints: Mango 14 document.write(fruits[fruits.length - 1]); // Prints: Papaya 15 </script> 16 </body> 17 </html> </pre>	<pre> Apple Banana Mango Papaya </pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------

Figure 75 – How to get individual array element by their index (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-arrays.php>)

Getting the Length of an Array

The length property returns the length of an array, which is the total number of elements included in the array. Array length is always greater than the index of any of its element.

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Get the Length of an Array</title>
6 </head>
7 <body>
8   <script>
9     var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"];
10    document.write(fruits.length); // Outputs: 5
11  </script>
12 </body>
13 </html>
```

Figure 76 – Getting the Length of an Array (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-arrays.php>)

Looping Through Array Elements

for loop can be used to gain access to each element of an array in sequential order, as follows:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Loop Through an Array Using For Loop</title>
6 </head>
7 <body>
8   <script>
9     var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"];
10
11    // Iterates over array elements
12    for(var i = 0; i < fruits.length; i++){
13      document.write(fruits[i] + "<br>"); // Print array element
14    }
15  </script>
16 </body>
17 </html>
```

Apple
Banana
Mango
Orange
Papaya

Figure 77 – Looping through array elements (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-arrays.php>)

ECMAScript 6 has introduced a simpler way to iterate over array element, which is for-of loop. In this loop there is no need to initialise and keep track of the loop counter variable (i).

Adding New Elements to an Array

To add a new element at the end of an array, simply use the push() method, as follows:

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Add a New Element at the End of an Array</title> 6 </head> 7 <body> 8 <script> 9 var colors = ["Red", "Green", "Blue"]; 10 colors.push("Yellow"); 11 12 document.write(colors + "
"); // Prints: Red,Green,Blue,Yellow 13 document.write(colors.length); // Prints: 4 14 </script> 15 </body> 16 </html> </pre>	<pre> Red,Green,Blue,Yellow 4 </pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------

Figure 78 – Adding New Elements to an Array (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-arrays.php>)

Likewise, to add a new element at the beginning of an array the unshift() method should be used, as follows:

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Add a New Element at the Beginning of an Array</title> 6 </head> 7 <body> 8 <script> 9 var colors = ["Red", "Green", "Blue"]; 10 colors.unshift("Yellow"); 11 12 document.write(colors + "
"); // Prints: Yellow,Red,Green,Blue 13 document.write(colors.length); // Prints: 4 14 </script> 15 </body> 16 </html> </pre>	<pre> Yellow,Red,Green,Blue 4 </pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------

Figure 79 – Add a new element at the beginning of an array using the unshift() method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-arrays.php>)

Removing Elements from an Array

To eliminate the last element from an array you can use the pop() method. This method returns the value that was popped out.

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Remove the Last Element from an Array</title>
6 </head>
7 <body>
8   <script>
9     var colors = ["Red", "Green", "Blue"];
10    var last = colors.pop();
11
12    document.write(last + "<br>"); // Prints: Blue
13    document.write(colors.length); // Prints: 2
14  </script>
15 </body>
16 </html>

```

Figure 80 – Removing Elements from an Array (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-arrays.php>)

Adding or Removing Elements at any Position

The splice() method is a very flexible array method that allows to add or remove elements from any index, using the syntax `arr.splice(startIndex, deleteCount, elem1, ..., elemN)`

This method takes three parameters: the first is the index at which to start splicing the array, it is required; the second is the number of elements to remove (0 should be used in case the programmer does not want to remove any elements), it is optional; and the third parameter is a set of replacement elements, it is also optional.

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Add or Remove Array Elements at any Index</title> 6 </head> 7 <body> 8 <script> 9 var colors = ["Red", "Green", "Blue"]; 10 var removed = colors.splice(0,1); // Remove the first element 11 12 document.write(colors + "
"); // Prints: Green,Blue 13 document.write(removed + "
"); // Prints: Red (one item array) 14 document.write(removed.length + "
"); // Prints: 1 15 16 removed = colors.splice(1, 0, "Pink", "Yellow"); // Insert two items at position one 17 document.write(colors + "
"); // Prints: Green,Pink,Yellow,Blue 18 document.write(removed + "
"); // Empty array 19 document.write(removed.length + "
"); // Prints: 0 20 21 removed = colors.splice(1, 1, "Purple", "Violet"); // Insert two values, remove one 22 document.write(colors + "
"); //Prints: Green,Purple,Violet,Yellow,Blue 23 document.write(removed + "
"); // Prints: Pink (one item array) 24 document.write(removed.length); // Prints: 1 25 </script> 26 </body> 27 </html> </pre>	<pre> Green,Blue Red 1 Green,Pink,Yellow,Blue 0 Green,Purple,Violet,Yellow,Blue Pink 1 </pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------

Figure 81 – Adding or Removing Elements at any Position (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-arrays.php>)

The splice() method returns an array of the deleted elements, or an empty array if no elements were deleted, as it can be seen in Figure 81. If the second argument is absent, all elements from the start to the end of the array are removed. Unlike slice() and concat() methods, the splice() method modifies the array on which it is called on.

Creating a String from an Array

There may be situations in which a programmer simply intends to create a string by joining the elements of an array. To do so, he/she can use the join() method. This method takes an optional parameter which is a separator string that is added in between each element. If you omit the separator, then JavaScript will use comma (,) by default.

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Join All Elements of an Array into a String</title> 6 </head> 7 <body> 8 <script> 9 var colors = ["Red", "Green", "Blue"]; 10 11 document.write(colors.join() + "
"); // Prints: Red,Green,Blue 12 document.write(colors.join("") + "
"); // Prints: RedGreenBlue 13 document.write(colors.join("-") + "
"); // Prints: Red-Green-Blue 14 document.write(colors.join(", ")); // Prints: Red, Green, Blue 15 </script> 16 </body> 17 </html> </pre>	<pre> Red,Green,Blue RedGreenBlue Red-Green-Blue Red, Green, Blue </pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------

Figure 81 – Creating a String from an Array (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-arrays.php>)

An array can also be converted to a comma-separated string using the `toString()`. This method does not allow the separator parameter as `join()`.

Merging Two or More Arrays

The `concat()` method can be used to combine two or more arrays. This method does not change the prevailing arrays, instead it returns a new array.

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Merge Two Arrays</title> 6 </head> 7 <body> 8 <script> 9 var pets = ["Cat", "Dog", "Parrot"]; 10 var wilds = ["Tiger", "Wolf", "Zebra"]; 11 12 // Creating new array by combining pets and wilds arrays 13 var animals = pets.concat(wilds); 14 document.write(animals); // Prints: Cat,Dog,Parrot,Tiger,Wolf,Zebra 15 </script> 16 </body> 17 </html> </pre>	<p>Cat,Dog,Parrot,Tiger,Wolf,Zebra</p>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------

Figure 82 – Merging Two or More Arrays (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-arrays.php>)

The `concat()` method can take any number of array arguments, so an array can be created from any number of other arrays, as shown in the following example:

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Merge Multiple Arrays</title> 6 </head> 7 <body> 8 <script> 9 var pets = ["Cat", "Dog", "Parrot"]; 10 var wilds = ["Tiger", "Wolf", "Zebra"]; 11 var bugs = ["Ant", "Bee"]; 12 13 // Creating new array by combining pets, wilds and bugs arrays 14 var animals = pets.concat(wilds, bugs); 15 document.write(animals); // Prints: Cat,Dog,Parrot,Tiger,Wolf,Zebra,Ant,Bee 16 </script> 17 </body> 18 </html> </pre>	<p>Cat,Dog,Parrot,Tiger,Wolf,Zebra,Ant,Bee</p>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------

Figure 83 – The `concat()` method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-arrays.php>)

Searching Through an Array

The `indexOf()` and `lastIndexOf()` methods can be used for searching an array for a specific value. If the value is found, both methods return an index representing the array element. If the value is not found, -1 is returned. The `indexOf()` method returns the first one found, although the `lastIndexOf()` returns the last one found. Both methods also recognise an optional integer parameter from index which specifies the index within the array at which to start the search.

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Search an Array for a Specific Value</title>
6 </head>
7 <body>
8   <script>
9     var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"];
10
11     document.write(fruits.indexOf("Apple") + "<br>"); // Prints: 0
12     document.write(fruits.indexOf("Banana") + "<br>"); // Prints: 1
13     document.write(fruits.indexOf("Pineapple")); // Prints: -1
14   </script>
15 </body>
16 </html>

```

Figure 84 – Searching through an array (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-arrays.php>)

`includes()` method can also be used to find out whether an array involves a certain element or not. This method takes the same parameters as `indexOf()` and `lastIndexOf()` methods, but it returns true or false instead of index number.

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JavaScript Find Whether an Array Includes a Certain Value</title>
6 </head>
7 <body>
8   <script>
9     var arr = [1, 0, 3, 1, false, 5, 1, 4, 7];
10
11     document.write(arr.includes(1) + "<br>"); // Prints: true
12     document.write(arr.includes(6) + "<br>"); // Prints: false
13     document.write(arr.includes(1, 2) + "<br>"); // Prints: true
14     document.write(arr.includes(3, 4)); // Prints: false
15   </script>
16 </body>
17 </html>

```

Figure 85 – Searching through an array using the `includes()` method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-arrays.php>)

To search an array based on certain condition the JavaScript `find()` method can be used, which has been recently introduced in ES6. This method returns the value of the first element in the array that fulfils the provided testing function. If not, it returns undefined. The `find()` method simply searches the first element that meets the provided testing function. Still, if the programmer intends to find out all the matched elements, the `filter()` method can be used. It creates a new array with all the elements that successfully passes the given test, as can be checked in [this example](#).

JavaScript Sorting Arrays

Sorting is a popular task when working with arrays. It can be used, for example, to display the city or country names in alphabetical order. The JavaScript Array object has a built-in method `sort()` for sorting array elements in alphabetical order.

<pre>1 <!DOCTYPE html> 2 <html lang="en"> 3 <head> 4 <meta charset="utf-8"> 5 <title>JavaScript Sort an Array Alphabetically</title> 6 </head> 7 <body> 8 <script> 9 var fruits = ["Banana", "Orange", "Apple", "Papaya", "Mango"]; 10 var sorted = fruits.sort(); 11 12 document.write(fruits + "
"); // Outputs: Apple,Banana,Mango,Orange,Papaya 13 document.write(sorted); // Outputs: Apple,Banana,Mango,Orange,Papaya 14 </script> 15 </body> 16 </html></pre>	<pre>Apple,Banana,Mango,Orange,Papaya Apple,Banana,Mango,Orange,Papaya</pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------

Figure 86 – The built-in method `sort()` for sorting array elements in alphabetical order (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-arrays.php>)

To **reverse the order of the elements** of an array, the `reverse()` method can be used. It reverses an array in such a way that the first array element becomes the last, and vice-versa. The `sort()` and `reverse()` method changes the initial array and return a reference to the same array, as it can be checked [here](#).

For **sorting numeric arrays**, it is not advisable to use the `sort()` method, as it can produce unexpected results. For this purpose, programmers should pass a compare

function, as when a compare function is specified, array elements are sorted according to the return value of the compare function.

For example, when comparing *a* and *b*:

- If the compare function returns a value less than 0, then *a* comes first.
- If the compare function returns a value greater than 0, then *b* comes first.
- If the compare function returns 0, *a* and *b* remain unchanged regarding each other, but grouped with respect to all other elements.

Therefore, since $5 - 20 = -15$ which is less than 0, therefore 5 comes first, likewise $20 - 10 = 10$ which is greater than 0, therefore 10 comes before 20, likewise $20 - 75 = -55$ which is less than 0, so 20 comes before 75, similarly 50 comes before 75, and so on. This can be checked in [this example](#).

To **find the maximum and minimum value in an Array**, the `apply()` method in combination with the `Math.max()` and `Math.min()` can be used, as follows in [this example](#). The `apply()` method offers an accessible way to pass array values as arguments to a function that accepts multiple arguments in an array-like manner, but not an array. Then, the resulting statement `Math.max.apply(null, numbers)` in the example above is equivalent to the `Math.max(3, -7, 10, 8, 15, 2)`.

Finally, to **sort an array of objects** the `sort()` method can be used. In [this example](#), it is shown how to sort an array of objects by property values.

JavaScript Loops

Loops are applied to perform the same block of code many times if a certain condition is encountered. The key idea behind a loop is to mechanize the repetitive tasks within a program to save the time and effort. JavaScript now supports five different types of loops:

- **while** — loops through a block of code if the condition specified evaluates to *true*.

- **do...while** — loops through a block of code one time; then the condition is evaluated. If the condition is *true*, the statement is repeated if the specified condition is *true*.
- **for** — loops through a block of code until the counter reaches a specified number.
- **for...in** — loops through the properties of an object.
- **for...of** — loops over iterable objects such as arrays, strings, etc.

The while loop

This is the easiest looping statement provided by JS. It loops through a block of code if the specified condition evaluates to *true*. Once the condition fails, the loop is stopped.

The generic syntax of the while loop is:

```
while(condition) {  
    // Code to be executed  
}
```

In [this example](#), a loop that keeps on running as long as the variable $i \leq 5$. As it can be checked, i will increase by 1 each time the loop runs. Programmers must ensure that the condition specified in the loop can eventually go false. If not, the loop will never stop iterating (infinite loop).

The do...while loop

The do-while loop is a variant of the while loop, which assesses the condition at the end of each loop iteration. With a do...while loop, the block of code is executed once, and then the condition is evaluated, if the condition is *true*, the statement is repeated if the specified condition evaluated to is *true*. Its common syntax is:


```
do {  
    // Code to be executed  
}  
while(condition);
```

The JavaScript code in [this example](#) defines a loop that starts with $i=1$. It will then produce the output and increase the value of variable i by 1. After that the condition is evaluated, and the loop will continue to run if $i \leq 5$.

The for loop

It repeats a block of code if a certain condition is met. It is normally used to implement a block of code for certain number of times. Its syntax is:

```
for(initialization; condition; increment) {  
    // Code to be executed  
}
```

The parameters of the **for loop** statement have the following implications:

- **initialisation** — it is used to set the counter variables and evaluated once entirely before the first execution of the body of the loop.
- **condition** — it is evaluated at the beginning of each iteration. If it evaluates to *true*, the loop statements achieve. If it evaluates to *false*, the execution of the loop ends.
- **increment** — it updates the loop counter with a new value each time the loop runs.

[This example](#) defines a loop that begins with $i=1$. The loop will go on until the value of $i \leq 5$. The variable i will increase by 1 each time the loop runs.

The for...in Loop

It is a special type of a loop that repeats over the properties of an object, or the elements of an array. Its syntax is:

```
for(variable in object) {  
    // Code to be executed  
}
```

The loop counter i.e., variable in the for-in loop is a string, not a number. It comprises the name of present property or the index of the current array element.

[This example](#) demonstrates how to loop through all properties of a JS object.

The for...of Loop

ES6 presents a new for-of loop which allows to iterate over arrays or other iterable objects (e.g., strings) very simply. Additionally, the code inside the loop is executed for each element of the iterable object. [This example](#) shows how to loop through arrays and strings using this loop.

JavaScript Functions

A function is a set of statements that accomplish specific tasks and can be kept and maintained independently. Functions give a way to create reusable code packages which are more manageable and easier to debug. Some advantages of using functions are as follows:

- **Functions decreases the repetition of code within a program** — Function allows to extract frequently used block of code into a single component. This way

it is possible to perform the same task by calling this function wherever needed within a script without having to copy and paste the same block of code over and over.

- **Functions makes the code much easier to maintain** — Since a function created once can be applied many times, any changes made inside a function are automatically implemented at all the places without affecting the respective files.
- **Functions makes it easier to get rid of the errors** — When the program is split into functions, if any error arise, the programmer knows precisely what function is causing the error and where to spot it. Consequently, fixing errors becomes much simpler.

Defining and Calling a Function

The declaration of a function starts with the function keyword, followed by the name of the function to be created, then followed by parentheses, and lastly by the function's code between curly brackets {}. To declare a function, the following syntax applies:

```
function functionName() {  
    // Code to be executed  
}
```

[This simple example](#) displays a “Hello” message.

Once a function is defined, it can be called from anywhere in the document, by typing its name followed by a set of parentheses, like sayHello() in the aforementioned example.

Adding Parameters to Functions

Parameters can be specified when defining a function to accept input values at run time. The parameters act like placeholder variables within a function; they are replaced at run time by the values (known as *argument*) offered to the function at the time of request.

Parameters are set on the first line of the function inside the set of parentheses, as follows:

```
function functionName(parameter1, parameter2, parameter3) {  
    // Code to be executed  
}
```

The displaySum() function in [this example](#) gets two numbers as arguments, simply add them as one and then display the result in the browser. There is no limit for defining parameters. Nevertheless, for each parameter specified, a corresponding argument requires to be passed to the function when it is called, if not its value becomes [undefined](#).

Default Values for Function Parameters

With ES6, it is possible to specify default values to the function parameters. This implies that if no arguments are provided to function when it is called, these default parameters values will be used. [This example](#) is quite explanative regarding how valuable this feature is, since to achieve the same result, the prior procedure was [this one](#).

Returning Values from a Function

A function can return a value back to the script that called the function as a result, by utilizing the return statement. The value may be of any type (i.e., arrays and objects). The return statement typically placed as the last line of the function before the closing curly bracket and ends it with a semicolon, as displayed [here](#).

A function can does not return multiple values. Still, similar results can be obtained by returning an array of values, as exhibited [here](#).

Working with Function Expressions

The syntax used before to create functions is called **function declaration**. There is another syntax for building a function – [function expression](#). Once stored in a variable, the [variable can be utilized as a function](#). The syntax of the function declaration and function expression seems very similar, but [they vary](#) in the way they are evaluated. As observed in the previous example, the function expression gave an exception when it was invoked before it is defined, but the function declaration executed effectively. JS analyses declaration function before the program executes. Hence, it does not make a difference if the program invokes the function before it is defined, as JavaScript has elevated the function to the top of the current scope in the background. The function expression is not assessed until it is assigned to a variable; consequently, it is still undefined when invoked.

Understanding the Variable Scope

Variables can be declared anywhere in JS. However, the location of the declaration will establish the extent of a variable's availability within the JavaScript program – this process is also called **variable scope**. By default, variables declared within a function have local scope, which means they cannot be viewed or controlled from outside of that function, as shown [here](#). Though, any variables declared in a program outside of a function has global scope, wherever that script is located concerning the function, as it can be checked [here](#).

JavaScript Objects

JavaScript is an object-based language and there nearly everything is an object or acts like one. Thus, to work with JS successfully and efficiently, programmers need to understand how objects work, as well as how to create objects and use them.

A JavaScript object is simply a collection of named values. They are typically referred to as properties of the object. Being an array a collection of values, in which each value has an index (a numeric key) starting from zero and incrementing by one for each value. An object is like an array, but the difference is that the programmer defines the keys, (name, age, gender, etc.).

Creating an object

Programmers can create objects with curly brackets, including a voluntary list of properties as well. A property can be “key:value” pair, in which the key (or *property name*) is always a string, and value (or *property value*) can be any data type (strings, numbers, Booleans, arrays, functions, etc.). Moreover, properties with functions as their values are frequently called methods to distinguish them from other features. A JS object may be as [this](#). This example creates an object called person, which has 3 properties (name, age and gender) and one method displayName(). This method shows the value of this.name, which agrees to person.name. This is the easiest and ideal way to create a new object in JF, which is known as **object literals syntax**. The property names generally do not need to be quoted except if they are reserved words, or if they contain spaces or special characters (anything other than letters, numbers, and the _ and \$ characters), or if they start with a number, as shown [here](#).

Accessing Object’s Properties

In order to access or get the value of a property, the dot (.) can be applied, as well as the square bracket ([]) notation, as [this example](#) shows. The dot notation is simpler to read and write, but it cannot always be used. If the name of the property is not valid (for

example, if it contains special characters or spaces), the dot notation cannot be used, but the [bracket notation instead](#). It presents much more flexibility than dot notation and additionally allows to identify property names as variables as a replacement for string literals.

Looping Through Object's Properties

Programmers can iterate through the key-value pairs of an object using the for...in loop. It is specifically enhanced for iterating over object's properties, as seen [here](#).

Setting Object's Properties

Likewise, new properties can be set, or the existing one can be updated by using the dot (.) or bracket ([]) notation, as demonstrated [here](#).

Deleting Object's Properties

The delete operator can be applied to entirely delete properties from an object. Removing is the only way to really get rid of a property. Adjusting the property to undefined or null simply changes the value of the property, it does not remove property from the object. So, it has no effect on variables or declared functions. Nevertheless, programmers should avoid delete operator for the purpose of deleting an array element, as it does not modify the array's length, it just drops a hole in the array.

Calling Object's Methods

An object's method can be accessed the same way as one would access properties—using the dot notation or applying the square bracket notation, as can be checked [here](#).

Manipulating by Value vs. Reference

JS objects are reference types, which means that when the programmer makes copies of them, they are simply copying the references to that object, while primitive values like strings and numbers are assigned or copied as a whole value. [This example](#) is quite

demonstrative of this idea. As it could be observed, a copy of a variable message has been done and it changed the value of that same copy. Both variables remain distinct and separate. However, if the same principle is applied to an object, a [different result will be gathered](#). So, any changes done to the variable user also interfere with the person variable, as both variables refer to the same object. Therefore, merely copying the object does not really clone it but copies the reference to that object.

5.2. JavaScript & DOM

What is the Document Object Model (DOM)?

DOM is created by the browser when a web page is loaded in HTML or XML documents. It is used to define the logical structure of these documents and to access and alter their elements.

The HTML DOM refers to the Document Object Model of the HTML documents, whereas the XML DOM refers to the Document Object Model of the XML documents. In this subchapter, we will focus on the HTML DOM which can be used to access and manipulate HTML documents via JavaScript.

The DOM is constructed as a hierarchical tree of objects that include all parts of an HTML document such as elements, attributes, text, etc. These individual objects of the tree are also known as nodes, and they are comprised of parent and child nodes. In graphical form, it will look something like the diagram provided below.

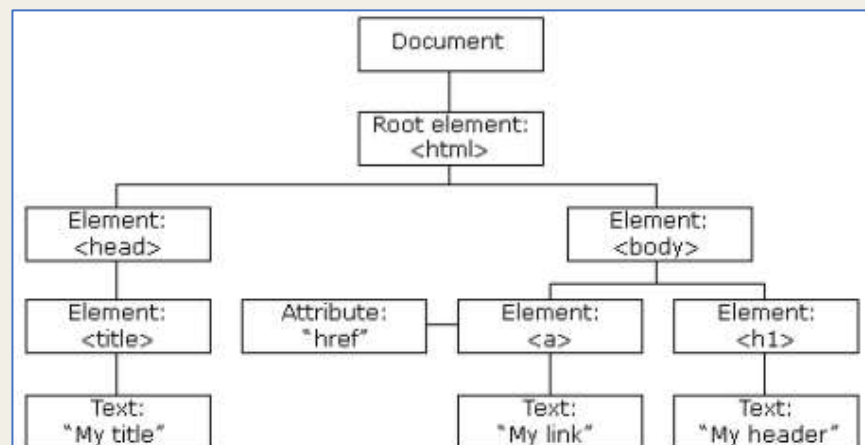


Figure 87 – HTML DOM Tree of Objects (Source: https://www.w3schools.com/js/js_htmlDOM.asp)

Within the HTML DOM, JavaScript can do the following:

- change all the HTML elements in the page
- change all the HTML attributes in the page
- change all the CSS styles in the page
- remove existing HTML elements and attributes
- add new HTML elements and attributes
- react to all existing HTML events in the page
- create new HTML events in the page

JavaScript DOM Selectors

Selecting DOM Elements in JavaScript

JavaScript is used to get or modify the content or value of the HTML elements of the web page and apply some special effects such as animations or hide. To be able to perform any action, you need to find or select the target HTML element.

We will go through some of the most common ways of selecting elements on a page and manipulating them with JavaScript.

Selecting the Topmost Elements

The topmost elements can be accessed directly as document properties.

For instance, to access the `<html>` element, use the `document.documentElement` property. For the `<head>` element, you can use the `document.head` property and for the `<body>` element, the `document.body` property.

Let's see an example to understand all of this a bit better.


```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JS Select Topmost Elements</title>
6 </head>
7 <body>
8   <script>
9     // Display lang attribute value of html element
10    alert(document.documentElement.getAttribute("lang")); // Outputs: en
11
12    // Set background color of body element
13    document.body.style.background = "yellow";
14
15    // Display tag name of the head element's first child
16    alert(document.head.firstChild.nodeName); // Outputs: meta
17  </script>
18 </body>
19 </html>

```

Figure 88 - Topmost Elements Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-selectors.php>)

* It is important to note that the `document.body` should not be used before the `<body>` element since it will return `null`. The program needs to go through the `<body>` element first to access the `document.body` property.

Let's look at this example to understand why the `<body>` will be `null`:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JS Document.body Demo</title>
6   <script>
7     alert("From HEAD: " + document.body); // Outputs: null (since <body> is not
8     // parsed yet)
9   </script>
10 </head>
11 <body>
12   <script>
13     alert("From BODY: " + document.body); // Outputs: HTMLBodyElement
14   </script>
15 </body>
16 </html>

```

Figure 89 - Topmost Elements Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-selectors.php>)

This example demonstrates what we saw in the beginning about the hierarchical relationships that exist between nodes. You need to be mindful that in order to access the document.body property, you will have to start from the <body> element to avoid null values.

Selecting Elements by ID

If you want to find or select an HTML element, the easiest way is to select it based on its unique ID. You can do this with the getElementById() method.

In the following example, the element that has an ID attribute id="mark" is selected and highlighted:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JS Select Element by ID</title>
6 </head>
7 <body>
8   <p id="mark">This is a paragraph of text.</p>
9   <p>This is another paragraph of text.</p>
10
11 <script>
12   // Selecting element with id mark
13   var match = document.getElementById("mark");
14
15   // Highlighting element's background
16   match.style.background = "yellow";
17 </script>
18 </body>
19 </html>

```

Figure 90 - Selecting Elements by ID Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-selectors.php>)

The getElementById() method is used to return the element as an object if a matching element is found. Otherwise, it will return null.

* Keep in mind that any HTML element can have an id attribute, which must be a **unique value within a page**. This essentially means that no two elements can have the same id.

Selecting Elements by Class Name

If you want to select all the elements with specific class names, use the `getElementsByClassName()` method. It will return an array-like object of all child elements which have all the given class names.

Let's look at an example:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JS Select Elements by Class Name</title>
6 </head>
7 <body>
8   <p class="test">This is a paragraph of text.</p>
9   <div class="block test">This is another paragraph of text.</div>
10  <p>This is one more paragraph of text.</p>
11
12  <script>
13    // Selecting elements with class test
14    var matches = document.getElementsByClassName("test");
15
16    // Displaying the selected elements count
17    document.write("Number of selected elements: " + matches.length);
18
19    // Applying bold style to first element in selection
20    matches[0].style.fontWeight = "bold";
21
22    // Applying italic style to last element in selection
23    matches[matches.length - 1].style.fontStyle = "italic";
24
25    // Highlighting each element's background through loop
26    for(var elem in matches) {
27      matches[elem].style.background = "yellow";
28    }
29  </script>
30 </body>
31 </html>

```

Figure 91 – Selecting Elements by Class Name Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-selectors.php>)

Selecting Elements by Tag Name

If you want to select elements by their tag name, use the `getElementsByTagName()` method. This method will also return an array-like object of all child elements which have the given tag name.

Let's look at an example to understand this a bit better:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JS Select Elements by Tag Name</title>
6 </head>
7 <body>
8   <p>This is a paragraph of text.</p>
9   <div class="test">This is another paragraph of text.</div>
10  <p>This is one more paragraph of text.</p>
11
12  <script>
13    // Selecting all paragraph elements
14    var matches = document.getElementsByTagName("p");
15
16    // Printing the number of selected paragraphs
17    document.write("Number of selected elements: " + matches.length);
18
19    // Highlighting each paragraph's background through loop
20    for(var elem in matches) {
21      matches[elem].style.background = "yellow";
22    }
23  </script>
24 </body>
25 </html>
```

Figure 92 - Selecting Elements by Tag Name Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-selectors.php>)

Selecting Elements with CSS Selectors

CSS Selectors offer a very powerful and efficient way to select HTML elements in a document. To select elements that match the specified CSS Selector, you can use the `querySelectorAll()` method.

This method will return a list of all the elements that match the specified selectors.

Follow the example below to see how you can examine this list like an array:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JS Select Elements with CSS Selectors</title>
6 </head>
7 <body>
8   <ul>
9     <li>Bread</li>
10    <li class="tick">Coffee</li>
11    <li>Pineapple Cake</li>
12  </ul>
13
14  <script>
15    // Selecting all li elements
16    var matches = document.querySelectorAll("ul li");
17
18    // Printing the number of selected li elements
19    document.write("Number of selected elements: " + matches.length + "<br>");
20
21    // Printing the content of selected li elements
22    for(var elem of matches) {
23      document.write(elem.innerHTML + "<br>");
24    };
25
26    // Applying line through style to first li element with class tick
27    matches = document.querySelectorAll("ul li.tick");
28    matches[0].style.textDecoration = "line-through";
29  </script>
30 </body>
31 </html>

```

Figure 93 - Selecting Elements with CSS Selectors Example (Source: <https://www.tutorialpublic.com/javascript-tutorial/javascript-dom-selectors.php>)

* Please note that the `querySelectorAll()` method supports CSS pseudo-classes such as `:first-child`, `:last-child`, `:hover`, etc. However, this method will always return an empty list for CSS pseudo-elements like `::before`, `::after`, `::first-line`, etc.

JavaScript DOM Styling

Styling DOM Elements in JavaScript

You can also change the visual presentation of HTML documents in a dynamic way by using JavaScript to apply different styles to HTML elements. Almost all element styles can be set such as fonts, colours, margins, borders, background images, text alignment, width and height, position, and so on.

Here, we will go through various methods that can be used to set styles in JavaScript.

Setting Inline Styles on Elements

The style attribute is used to apply inline styles directly to the specific HTML element. The `style` property is used in JavaScript to get or set the inline style of an element.

In the following example, colour and font properties will be set for an element with `id="intro"`:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JS Set Inline Styles Demo</title>
6 </head>
7 <body>
8   <p id="intro">This is a paragraph.</p>
9   <p>This is another paragraph.</p>
10
11 <script>
12   // Selecting element
13   var elem = document.getElementById("intro");
14
15   // Applying styles on element
16   elem.style.color = "blue";
17   elem.style.fontSize = "18px";
18   elem.style.fontWeight = "bold";
19 </script>
20 </body>
21 </html>
```

Figure 94 - Inline Styles on Elements Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-styling.php>)

Naming Conventions of CSS Properties in JavaScript

It is important to mention that many of CSS properties contain hyphens (-) in their names such as font-size, background-image, text-decoration, etc. However, in JavaScript, the hyphen is a reserved operator that signifies a minus sign. Therefore, it is not possible to write an expression in this way: elem.style.font-size.

To overcome this issue, CSS property names in JavaScript that contain one or more hyphens are converted to intercapitalised style words. This essentially means that the hyphens are removed and the first letter after the hyphen is capitalised. For instance, the CSS property font-size become fontSize in DOM property.

Getting Style Information from Elements

The style property is also used to get the styles applied to HTML elements.

The example below will get style information from the element with id="intro":

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>JS Get Element's Style Demo</title>
6 </head>
7 <body>
8   <p id="intro" style="color:red; font-size:28px;">This is a paragraph.</p>
9   <p>This is another paragraph.</p>
10
11   <script>
12     // Selecting element
13     var elem = document.getElementById("intro");
14
15     // Getting style information from element
16     alert(elem.style.color); // Outputs: red
17     alert(elem.style.fontSize); // Outputs: 28px
18     alert(elem.style.fontStyle); // Outputs nothing
19   </script>
20 </body>
21 </html>
```

Figure 95 - Style property – Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-styling.php>)

The style property isn't the most useful when it comes to getting style information from the elements since it only returns the style rules that are set in the element's style attribute and not those that come from elsewhere such as style rules in the embedded style sheets, or external style sheets.

If you want to get the values of all CSS properties that are used to render an element you can use the `window.getComputedStyle()` method, as shown in the following example:

```

1 <!DOCTYPE HTML>
2 <html>
3 <head>
4 <meta charset="utf-8">
5 <title>JS Get Computed Style Demo</title>
6 <style type="text/css">
7   #intro {
8     font-weight: bold;
9     font-style: italic;
10  }
11 </style>
12 </head>
13 <body>
14   <p id="intro" style="color:red; font-size:20px;">This is a paragraph.</p>
15   <p>This is another paragraph.</p>
16
17   <script>
18     // Selecting element
19     var elem = document.getElementById("intro");
20
21     // Getting computed style information
22     var styles = window.getComputedStyle(elem);
23
24     alert(styles.getPropertyValue("color")); // Outputs: rgb(255, 0, 0)
25     alert(styles.getPropertyValue("font-size")); // Outputs: 20px
26     alert(styles.getPropertyValue("font-weight")); // Outputs: 700
27     alert(styles.getPropertyValue("font-style")); // Outputs: italic
28   </script>
29 </body>
30 </html>

```

Figure 96 - `window.getComputedStyle()` – Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-styling.php>)

* Keep in mind that the value 700 for the CSS property font-weight is the same as the keyword bold. The colour keyword red is the same as `rgb(255,0,0)`, which is the `rgb` notation of a colour.

Adding CSS Classes to Elements

Another way to get or set CSS classes to HTML elements is by using the `className` property. Class is a reserved word in JavaScript; thus, JavaScript uses the `className` property to refer to the value of the HTML class attribute.

Let's look at the following example to learn how to add a new class, or replace all existing classes to a `<div>` element with `id="info"`:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="utf-8">
5 <title>JS Add or Replace CSS Classes Demo</title>
6 <style>
7   .highlight {
8     background: yellow;
9   }
10 </style>
11 </head>
12 <body>
13   <div id="info" class="disabled">Something very important!</div>
14
15   <script>
16     // Selecting element
17     var elem = document.getElementById("info");
18
19     elem.className = "note"; // Add or replace all classes with note class
20     elem.className += " highlight"; // Add a new class highlight
21   </script>
22 </body>
23 </html>

```

Figure 97 - Adding Classes to Elements Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-styling.php>)

An even better way to work with CSS classes is by using the `classList` property to get, set or remove CSS classes easily from an element. This property is supported in all major browsers except Internet Explorer before version 10.

Let's see an example of this property:


```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="utf-8">
5 <title>JS classList Demo</title>
6 <style>
7     .highlight {
8         background: yellow;
9     }
10 </style>
11 </head>
12 <body>
13     <div id="info" class="disabled">Something very important!</div>
14
15     <script>
16         // Selecting element
17         var elem = document.getElementById("info");
18
19         elem.classList.add("hide"); // Add a new class
20         elem.classList.add("note", "highlight"); // Add multiple classes
21         elem.classList.remove("hide"); // Remove a class
22         elem.classList.remove("disabled", "note"); // Remove multiple classes
23         elem.classList.toggle("visible"); // If class exists remove it, if not add it
24
25         // Determine if class exist
26         if(elem.classList.contains("highlight")) {
27             alert("The specified class exists on the element.");
28         }
29     </script>
30 </body>
31 </html>

```

Figure 98 - classList property - Adding Classes to Elements Example (**Source:** <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-styling.php>)

JavaScript DOM Get Set Attributes

Working with Attributes

Attributes are special words used inside the start tag of an HTML element to control the tag's behaviour or provide more information about the tag.

In this section, we will go through several methods of adding, removing or changing an HTML element's attribute.

Getting Element's Attribute Value

To get the current value of an element's attribute, you can use the `getAttribute()` method. If that particular attribute is not found on the element, it will return null.

Let's see the following example below:

```
1 <a href="https://www.google.com/" target="_blank" id="myLink">Google</a>
2
3 <script>
4     // Selecting the element by ID attribute
5     var link = document.getElementById("myLink");
6
7     // Getting the attributes values
8     var href = link.getAttribute("href");
9     alert(href); // Outputs: https://www.google.com/
10
11    var target = link.getAttribute("target");
12    alert(target); // Outputs: _blank
13 </script>
```

Figure 99 - `getAttribute()` method – Getting Element's Attribute Value Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-get-set-attributes.php>)

Setting Attributes on Elements

If you want to set an attribute on a specified element, you can use the `setAttribute()` method. If the attribute already exists on the element, the value will be updated. If not, a new attribute will be added with specified name and value.

In the following example, we will add a class and a disabled attribute to the `<button>` element:

```

1 <button type="button" id="myBtn">Click Me</button>
2
3 <script>
4     // Selecting the element
5     var btn = document.getElementById("myBtn");
6
7     // Setting new attributes
8     btn.setAttribute("class", "click-btn");
9     btn.setAttribute("disabled", "");
10 </script>

```

Figure 100 - `setAttribute()` method – Setting Attributes on Elements Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-get-set-attributes.php>)

If you want to update or change the value of an existing attribute on an element, you can also use the `setAttribute()` method.

Let's see an example that will update the value of the existing href attribute of an anchor (`<a>`) element:

```

1 <a href="#" id="myLink">Tutorial Republic</a>
2
3 <script>
4     // Selecting the element
5     var link = document.getElementById("myLink");
6
7     // Changing the href attribute value
8     link.setAttribute("href", "https://www.tutorialrepublic.com");
9 </script>

```

Figure 101 - `setAttribute()` method – Updating or Changing Attributes on Elements Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-get-set-attributes.php>)

Removing Attributes from Elements

To remove an attribute from a specific element, you can use the `removeAttribute()` method.

Remember the href attribute that we changed from the anchor element; we are now going to remove it in the following example:

```
1 <a href="https://www.google.com/" id="myLink">Google</a>
2
3 <script>
4     // Selecting the element
5     var link = document.getElementById("myLink");
6
7     // Removing the href attribute
8     link.removeAttribute("href");
9 </script>
```

Figure 102 - removeAttribute() method – Removing Attributes from Elements Example

(Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-get-set-attributes.php>)

JavaScript DOM Manipulation

Manipulating DOM Elements in JavaScript

So far, we have learnt how to select and style HTML DOM elements. Now, we will learn how to add or remove DOM elements in a dynamic way, how to get their contents and many more.

Adding New Elements to DOM

The `document.createElement()` method is used to create a new element in an HTML document. It creates a new element; however, it does not add it to the DOM.

A separate step is needed to add it to the DOM, as shown in the example below:

```
1 <div id="main">
2   <h1 id="title">Hello World!</h1>
3   <p id="hint">This is a simple paragraph.</p>
4 </div>
5
6 <script>
7 // Creating a new div element
8 var newDiv = document.createElement("div");
9
10 // Creating a text node
11 var newContent = document.createTextNode("Hi, how are you doing?");
12
13 // Adding the text node to the newly created div
14 newDiv.appendChild(newContent);
15
16 // Adding the newly created element and its content into the DOM
17 var currentDiv = document.getElementById("main");
18 document.body.appendChild(newDiv, currentDiv);
19 </script>
```

Figure 103 - Adding New Elements Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-manipulation.php>)

In the example we just saw, the `appendChild()` is used to add the new element at the end of any other children under the specified parent node.

You also have the option to add the new element before any other children, as shown in the example below:

```
1 <div id="main">
2   <h1 id="title">Hello World!</h1>
3   <p id="hint">This is a simple paragraph.</p>
4 </div>
5
6 <script>
7 // Creating a new div element
8 var newDiv = document.createElement("div");
9
10 // Creating a text node
11 var newContent = document.createTextNode("Hi, how are you doing?");
12
13 // Adding the text node to the newly created div
14 newDiv.appendChild(newContent);
15
16 // Adding the newly created element and its content into the DOM
17 var currentDiv = document.getElementById("main");
18 document.body.insertBefore(newDiv, currentDiv);
19 </script>
```

Figure 104 - Adding New Elements Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-manipulation.php>)

Getting or Setting HTML Contents to DOM

If you want to get or set the contents of HTML elements, you can use the innerHTML property. This property is used to set or get the HTML markup inside the element, which contains content between its opening and closing tags.

Let's look at an example to get a better understanding of this:


```
1 <div id="main">
2   <h1 id="title">Hello World!</h1>
3   <p id="hint">This is a simple paragraph.</p>
4 </div>
5
6 <script>
7 // Getting inner HTML contents
8 var contents = document.getElementById("main").innerHTML;
9 alert(contents); // Outputs inner html contents
10
11 // Setting inner HTML contents
12 var mainDiv = document.getElementById("main");
13 mainDiv.innerHTML = "<p>This is <em>newly inserted</em> paragraph.</p>";
14 </script>
```

Figure 105 - Getting or Setting HTML Contents to DOM Example (Source:

<https://www.tutorialpublic.com/javascript-tutorial/javascript-dom-manipulation.php>)

As you can from the example, new elements are inserted quite easily into the DOM with the innerHTML property. But this property replaces all the existing content of an element.

Therefore, if you do not want to replace the existing contents of an element, you can use the insertAdjacentHTML() method. This method takes two parameters: the HTML to be inserted and its position. The position must be one of the following: "beforebegin", "afterbegin", "beforeend", and "afterend". It is also significant to note that this method is supported in all major browsers.

In the following example, you can see how the positioning works:

```

1 <!-- beforebegin -->
2 <div id="main">
3   <!-- afterbegin -->
4   <h1 id="title">Hello World!</h1>
5   <!-- beforeend -->
6 </div>
7 <!-- afterend -->
8
9 <script>
10 // Selecting target element
11 var mainDiv = document.getElementById("main");
12
13 // Inserting HTML just before the element itself, as a previous sibling
14 mainDiv.insertAdjacentHTML('beforebegin', '<p>This is paragraph one.</p>');
15
16 // Inserting HTML just inside the element, before its first child
17 mainDiv.insertAdjacentHTML('afterbegin', '<p>This is paragraph two.</p>');
18
19 // Inserting HTML just inside the element, after its last child
20 mainDiv.insertAdjacentHTML('beforeend', '<p>This is paragraph three.</p>');
21
22 // Inserting HTML just after the element itself, as a next sibling
23 mainDiv.insertAdjacentHTML('afterend', '<p>This is paragraph four.</p>');
24 </script>

```

Figure 106 - insertAdjacentHTML() method - Getting or Setting HTML Contents to DOM Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-manipulation.php>)

- * Keep in mind that for the beforebegin and afterend positions to work the node has to be in the DOM tree and have a parent element.
- * Also, when inserting an HTML into a page, be careful not to use user input that hasn't been escaped/sanitised, to prevent XSS attacks.

Removing Existing Elements from DOM

To remove a child node from the DOM, you can use the removeChild() method. This method will also return the removed node.

Let's see the example below:

```
1 <div id="main">
2   <h1 id="title">Hello World!</h1>
3   <p id="hint">This is a simple paragraph.</p>
4 </div>
5
6 <script>
7   var parentElem = document.getElementById("main");
8   var childElem = document.getElementById("hint");
9   parentElem.removeChild(childElem);
10 </script>
```

Figure 107 - Removing Existing Elements from DOM Example (Source:

<https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-manipulation.php>)

You can also remove the child element without knowing the parent element. You can find the child element and use the parentNode property to find its parent. It will return the parent of the given node in the DOM tree.

```
1 <div id="main">
2   <h1 id="title">Hello World!</h1>
3   <p id="hint">This is a simple paragraph.</p>
4 </div>
5
6 <script>
7   var childElem = document.getElementById("hint");
8   childElem.parentNode.removeChild(childElem);
9 </script>
```

Figure 108 - Removing Existing Elements from DOM Example (Source:

<https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-manipulation.php>)

Replacing Existing Elements in DOM

You also have the option of replacing an element in HTML DOM with another by using the `replaceChild()` method. This method takes on two parameters: the node to be inserted and the node to be replaced. The syntax is used is as follows: `parentNode.replaceChild(newChild, oldChild);`

```
1 <div id="main">
2   <h1 id="title">Hello World!</h1>
3   <p id="hint">This is a simple paragraph.</p>
4 </div>
5
6 <script>
7 var parentElem = document.getElementById("main");
8 var oldPara = document.getElementById("hint");
9
10 // Creating new element
11 var newPara = document.createElement("p");
12 var newContent = document.createTextNode("This is a new paragraph.");
13 newPara.appendChild(newContent);
14
15 // Replacing old paragraph with newly created paragraph
16 parentElem.replaceChild(newPara, oldPara);
17 </script>
```

Figure 109 - Replacing Existing Elements in DOM Example (Source:

<https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-manipulation.php>)

JavaScript DOM Navigation

Navigating Between DOM Nodes

By now, you should have a better idea of how to select individual elements on a web page. There are many occasions where you would need to access child, parent or ancestor element. We have talked about nodes in the beginning of this subchapter and now we will see how we can access the different types of nodes.

DOM nodes have several properties and methods that let you navigate or traverse through the tree DOM structure and make necessary changes quite easily.

Accessing the Child Nodes

The `firstChild` and `lastChild` properties allow you to access the first and last direct child node of a node respectively. If a node does not have any child element, it will return null.

Let's check out the example below:

```
1 <div id="main">
2   <h1 id="title">My Heading</h1>
3   <p id="hint"><span>This is some text.</span></p>
4 </div>
5
6 <script>
7 var main = document.getElementById("main");
8 console.log(main.firstChild.nodeName); // Prints: #text
9
10 var hint = document.getElementById("hint");
11 console.log(hint.firstChild.nodeName); // Prints: SPAN
12 </script>
```

Figure 110 - Accessing Child Nodes Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-navigation.php>)

* Please note that the `nodeName` is a read-only property, which returns the name of the current node as a string. For example, it will return the tag name of an element node, `#text` for text node, `#comment` for comment node, `#document` for document node, and so on.

In the example that we just saw, the `nodeName` of the first child node of the main DIV elements returned `#text` instead of `H1`. This happens because white space, i.e., spaces, tabs, newlines, and so on, are considered valid characters and they become part of the

DOM tree in the form of #text nodes. Then, the <div> tag that contains a newline before the <h1> will create #text node.

In order to prevent this issue with the firstChild and lastChild returning #text or #comment nodes, you can use the firstElementChild and lastElementChild properties as an alternative. These properties will return only the first and last element of the node respectively. However, this will not work in Internet Explorer prior to Version 9.

The example below will give you a better understanding of this:

```
1 <div id="main">
2   <h1 id="title">My Heading</h1>
3   <p id="hint"><span>This is some text.</span></p>
4 </div>
5
6 <script>
7 var main = document.getElementById("main");
8 alert(main.firstElementChild.nodeName); // Outputs: H1
9 main.firstElementChild.style.color = "red";
10
11 var hint = document.getElementById("hint");
12 alert(hint.firstElementChild.nodeName); // Outputs: SPAN
13 hint.firstElementChild.style.color = "blue";
14 </script>
```

Figure 111 - Accessing Child Nodes Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-navigation.php>)

To access all child nodes of a given element, you can also use the childNodes property. Keep in mind that the first child node is assigned index 0.

See the example provided below:

```
1 <div id="main">
2   <h1 id="title">My Heading</h1>
3   <p id="hint"><span>This is some text.</span></p>
4 </div>
5
6 <script>
7   var main = document.getElementById("main");
8
9   // First check that the element has child nodes
10  if(main.childNodes()) {
11    var nodes = main.childNodes;
12
13    // Loop through node list and display node name
14    for(var i = 0; i < nodes.length; i++) {
15      alert(nodes[i].nodeName);
16    }
17  }
18 </script>
```

Figure 112 - Accessing Child Nodes Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-navigation.php>)

Here, the `childNodes` returns *all child nodes*, including non-element nodes like text and comment nodes.

If you want to get a *collection of only elements*, you should use the `children` property instead.

Let's look at an example to understand how to use this property:

```
1 <div id="main">
2   <h1 id="title">My Heading</h1>
3   <p id="hint"><span>This is some text.</span></p>
4 </div>
5
6 <script>
7   var main = document.getElementById("main");
8
9   // First check that the element has child nodes
10  if(main.hasChildNodes()) {
11    var nodes = main.children;
12
13    // Loop through node list and display node name
14    for(var i = 0; i < nodes.length; i++) {
15      alert(nodes[i].nodeName);
16    }
17  }
18 </script>
```

Figure 113 - Accessing Child Nodes Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-navigation.php>)

Accessing the Parent Nodes

To access the parent node of a specific node in the DOM tree, you can use the `parentNode` property.

* Note that the `parentNode` property will always return null values for document nodes because they do not have parents.

In the following example, you can see how the `parentNode` property is used:

```

1 <div id="main">
2   <h1 id="title">My Heading</h1>
3   <p id="hint"><span>This is some text.</span></p>
4 </div>
5
6 <script>
7 var hint = document.getElementById("hint");
8 alert(hint.parentNode.nodeName); // Outputs: DIV
9 alert(document.documentElement.parentNode.nodeName); // Outputs: #document
10 alert(document.parentNode); // Outputs: null
11 </script>

```

Figure 114 - Accessing Parent Nodes Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-navigation.php>)

It is good to know that the topmost DOM tree nodes can be accessed directly as document properties. We saw some examples of the topmost DOM tree nodes in earlier sections such as the <html> element, which can be accessed with document.documentElement property. Also, the <head> element can be accessed with document.head property, and the <body> element can be accessed with document.body property.

There is also an option to get only element nodes with the parentNode, as shown in the example below:

```

1 <div id="main">
2   <h1 id="title">My Heading</h1>
3   <p id="hint"><span>This is some text.</span></p>
4 </div>
5
6 <script>
7 var hint = document.getElementById("hint");
8 alert(hint.parentNode.nodeName); // Outputs: DIV
9 hint.parentNode.style.backgroundColor = "yellow";
10 </script>

```

Figure 115 - Accessing Parent Nodes Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-navigation.php>)

Accessing the Sibling Nodes

To access the previous and next node in the DOM tree, you can use the `previousSibling` and `nextSibling` properties respectively.

Let's look at an example:

```
1 <div id="main">
2   <h1 id="title">My Heading</h1>
3   <p id="hint"><span>This is some text.</span></p><hr>
4 </div>
5
6 <script>
7 var title = document.getElementById("title");
8 alert(title.previousSibling.nodeName); // Outputs: #text
9
10 var hint = document.getElementById("hint");
11 alert(hint.nextSibling.nodeName); // Outputs: HR
12 </script>
```

Figure 116 - Accessing Sibling Nodes Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-navigation.php>)

To skip any whitespace text nodes, you can use the `previousElementSibling` and `nextElementSibling` as alternatives to get the previous and next sibling elements. If no such sibling is found, these properties will return null values.

Let's see an example below:


```

1 <div id="main">
2   <h1 id="title">My Heading</h1>
3   <p id="hint"><span>This is some text.</span></p>
4 </div>
5
6 <script>
7 var hint = document.getElementById("hint");
8 alert(hint.previousElementSibling.nodeName); // Outputs: H1
9 alert(hint.previousElementSibling.textContent); // Outputs: My Heading
10
11 var title = document.getElementById("title");
12 alert(title.nextElementSibling.nodeName); // Outputs: P
13 alert(title.nextElementSibling.textContent); // Outputs: This is some text.
14 </script>

```

Figure 117 - Accessing Sibling Nodes Example (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-navigation.php>)

The `textContent` property used here signifies the text content of a node and all of its descendants.

Types of DOM Nodes

The DOM tree is comprised of different types of nodes that includes elements, text, comments and many more.

Every node has a `nodeType` property that can help you understand how you can access and manipulate said node. The following table provides a list with the most important and used node types that one needs to know:

Constant	Value	Description
ELEMENT_NODE	1	An element node such as <code><p></code> or <code></code> .
TEXT_NODE	3	The actual text of element.
COMMENT_NODE	8	A comment node i.e. <code><!-- some comment --></code>
DOCUMENT_NODE	9	A document node i.e. the parent of <code><html></code> element.
DOCUMENT_TYPE_NODE	10	A document type node e.g. <code><!DOCTYPE html></code> for HTML5 documents.

Figure 118 - Table of Most Common Types of DOM Nodes (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dom-navigation.php>)

5.3. JavaScript & BOM

JavaScript Window - The Browser Object Model

The Browser Object Model (BOM) allows JavaScript to "talk to" the browser.

The Browser Object Model (BOM)

There are no official standards for the Browser Object Model (BOM).

Since modern browsers have implemented (almost) the same methods and properties for JavaScript interactivity, it is often referred to, as methods and properties of the BOM.

The Window Object

The window object is supported by all browsers. It represents the browser's window.

All global JavaScript objects, functions, and variables automatically become members of the window object.

Global variables are properties of the window object.

Global functions are methods of the window object.

Even the document object (of the HTML DOM) is a property of the window object:

```
window.document.getElementById("header");
```

Figure 119 – The Window Object on JS BOM (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window.php>)

Is the same as:

```
document.getElementById("header");
```

Figure 120 – The Window Object on JS BOM – alternative version (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window.php>)

Window Size

Two properties can be used to determine the size of the browser window.

Both properties return the sizes in pixels:

- `window.innerHeight` - the inner height of the browser window (in pixels)
- `window.innerWidth` - the inner width of the browser window (in pixels)

```
let w = window.innerWidth;  
let h = window.innerHeight;
```

Figure 121 – The Window size alternatives (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window.php>)

Other Window Methods

Some other methods:

- `window.open()` - open a new window
- `window.close()` - close the current window
- `window.moveTo()` - move the current window
- `window.resizeTo()` - resize the current window

JavaScript Window Screen

The window.screen object contains information about the user's screen.

The window.screen object can be written without the window prefix.

Properties:

- screen.width
- screen.height
- screen.availWidth
- screen.availHeight
- screen.colorDepth
- screen.pixelDepth

Window Screen Width

The screen.width property returns the width of the visitor's screen in pixels.

Example

Display the width of the screen in pixels:

```
document.getElementById("demo").innerHTML =  
"Screen Width: " + screen.width;
```

Result will be:

```
Screen Width: 1280
```

Figure 122 – The screen.width property (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window.php>)

Window Screen Height

The screen.height property returns the height of the visitor's screen in pixels.

Example

Display the height of the screen in pixels:

```
document.getElementById("demo").innerHTML =  
"Screen Height: " + screen.height;
```

Result will be:

```
Screen Height: 720
```

Figure 123 – The Window `screen.height` property (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window.php>)

Window Screen Available Width

The `screen.availWidth` property returns the width of the visitor's screen, in pixels, minus interface features like the Windows Taskbar.

Example

Display the available width of the screen in pixels:

```
document.getElementById("demo").innerHTML =  
"Available Screen Width: " + screen.availWidth;
```

Result will be:

```
Available Screen Width: 1280
```

Figure 124 – The Window `screen.availWidth` property (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window.php>)

Window Screen Available Height

The `screen.availHeight` property returns the height of the visitor's screen, in pixels, minus interface features like the Windows Taskbar.

Example

Display the available height of the screen in pixels:

```
document.getElementById("demo").innerHTML =  
"Available Screen Height: " + screen.availHeight;
```

Result will be:

```
Available Screen Height: 680
```

Figure 125 – The Window `screen.availWidth` property (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window.php>)

Window Screen Colour Depth

The `screen.colorDepth` property returns the number of bits used to display one colour.

All modern computers use 24 bit or 32-bit hardware for colour resolution:

24 bits = 16,777,216 different "True Colours"

32 bits = 4,294,967,296 different "Deep Colours"

Older computers used 16 bits: 65,536 different "High Colours" resolution.

Very old computers, and old cell phones used 8 bits: 256 different "VGA colours".

Example

Display the color depth of the screen in bits:

```
document.getElementById("demo").innerHTML =  
"Screen Color Depth: " + screen.colorDepth;
```

Result will be:

```
Screen Color Depth: 24
```

Figure 126 – The Window `screen.availWidth` property (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window.php>)

Window Screen Pixel Depth

The `screen.pixelDepth` property returns the pixel depth of the screen.

Example

Display the pixel depth of the screen in bits:

```
document.getElementById("demo").innerHTML =  
"Screen Pixel Depth: " + screen.pixelDepth;
```

Result will be:

```
Screen Pixel Depth: 24
```

Figure 127 – The Window `screen.pixelDepth` property (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window.php>)

JavaScript Window Location

The `window.location` object can be used to get the current page address (URL) and to redirect the browser to a new page.

Window Location

The `window.location` object can be written without the `window` prefix.

Some examples:

- `window.location.href` returns the href (URL) of the current page
- `window.location.hostname` returns the domain name of the web host
- `window.location.pathname` returns the path and filename of the current page
- `window.location.protocol` returns the web protocol used (`http:` or `https:`)
- `window.location.assign()` loads a new document

Window Location Href

The `window.location.href` property returns the URL of the current page.

Example

Display the href (URL) of the current page:

```
document.getElementById("demo").innerHTML =  
"Page location is " + window.location.href;
```

Result is:

```
Page location is https://www.w3schools.com/js/js_window_location.asp
```

Figure 128 – The Window `window.location.href` property (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window-location.php>)

Window Location Hostname

The `window.location.hostname` property returns the name of the internet host (of the current page).

Example

Display the name of the host:

```
document.getElementById("demo").innerHTML =  
"Page hostname is " + window.location.hostname;
```

Result is:

```
Page hostname is www.w3schools.com
```

Figure 129 – The Window `window.location.hostname` property (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window-location.php>)

Window Location Pathname

The `window.location.pathname` property returns the pathname of the current page.

Example

Display the path name of the current URL:

```
document.getElementById("demo").innerHTML =  
"Page path is " + window.location.pathname;
```

Result is:

```
Page path is /js/js_window_location.asp
```

Figure 130 – The Window `window.location.pathname` property (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window-location.php>)

Window Location Protocol

The `window.location.protocol` property returns the web protocol of the page.

Example

Display the web protocol:

```
document.getElementById("demo").innerHTML =  
"Page protocol is " + window.location.protocol;
```

Result is:

```
Page protocol is https:
```

Figure 131 – The Window `window.location.protocol` property (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window-location.php>)

Window Location Port

The `window.location.port` property returns the number of the internet host port (of the current page).

Example

Display the name of the host:

```
document.getElementById("demo").innerHTML =  
"Port number is " + window.location.port;
```

Result is:

```
Port number is
```

Figure 132 – The Window `window.location.port` property (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window-location.php>)

Window Location Assign

The `window.location.assign()` method loads a new document.

```
Example
Load a new document:

<html>
<head>
<script>
function newDoc() {
  window.location.assign("https://www.w3schools.com")
}
</script>
</head>
<body>

<input type="button" value="Load new document" onclick="newDoc()">

</body>
</html>
```

Figure 133 – The Window `window.location.assign()` property (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window-location.php>)

JavaScript Window History

The `window.history` object contains the browsers history.

Window History

The `window.history` object can be written without the `window` prefix.

To protect the privacy of the users, there are limitations to how JavaScript can access this object.

Some methods:

- `history.back()` - same as clicking back in the browser
- `history.forward()` - same as clicking forward in the browser

Window History Back

The `history.back()` method loads the previous URL in the history list.

This is the same as clicking the Back button in the browser.

Example

Create a back button on a page:

```
<html>
<head>
<script>
function goBack() {
  window.history.back()
}
</script>
</head>
<body>

<input type="button" value="Back" onclick="goBack()">

</body>
</html>
```

The output of the code above will be:

Back

Figure 134 – The Window `history.back()` property (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window-location.php>)

Window History Forward

The `history.forward()` method loads the next URL in the history list.

This is the same as clicking the Forward button in the browser.

Example

Create a forward button on a page:

```
<html>
<head>
<script>
function goForward() {
  window.history.forward()
}
</script>
</head>
<body>

<input type="button" value="Forward" onclick="goForward()">

</body>
</html>
```

The output of the code above will be:

Forward

Figure 135 – The Window `history.forward()` property (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window-location.php>)

JavaScript Window Navigator

The window.navigator object contains information about the visitor's browser.

Window Navigator

The window.navigator object can be written without the window prefix.

Some examples:

- navigator.appName
- navigator.appCodeName
- navigator.platform

Browser Cookies

The cookieEnabled property returns true if cookies are enabled, otherwise false:

```
document.getElementById("demo").innerHTML =  
"cookiesEnabled is " + navigator.cookieEnabled;
```

Figure 136 – The Window history.forward() property (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-window-location.php>)

Browser Application Name

The appName property returns the application name of the browser.

Browser Application Code Name

The appCodeName property returns the application code name of the browser.

The Browser Engine

The product property returns the product name of the browser engine.

The Browser Version

The `appVersion` property returns version information about the browser.

The Browser Agent

The `userAgent` property returns the user-agent header sent by the browser to the server.

Warning !!!

The information from the navigator object can often be misleading, and should not be used to detect browser versions because:

- Different browsers can use the same name
- The navigator data can be changed by the browser owner
- Some browsers misidentify themselves to bypass site tests
- Browsers cannot report new operating systems, released later than the browser

The Browser Platform

The `platform` property returns the browser platform (operating system).

The Browser Language

The `language` property returns the browser's language.

Is The Browser Online?

The `onLine` property returns true if the browser is online.

Is Java Enabled?

The `javaEnabled()` method returns true if Java is enabled.

JavaScript Popup Boxes

JavaScript has three kind of popup boxes: Alert box, Confirm box, and Prompt box.

Alert Box

An alert box is often used if you want to make sure information comes through to the user.

When an alert box pops up, the user will have to click "OK" to proceed.

Syntax:

```
window.alert("sometext");
```

Figure 137 – Alert box syntax (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dialog-boxes.php>)

The `window.alert()` method can be written without the `window` prefix.

```
alert("I am an alert box!");
```

Figure 138 – The `window.alert()` method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dialog-boxes.php>)

Confirm Box

A confirm box is often used if you want the user to verify or accept something.

When a confirm box pops up, the user will have to click either "OK" or "Cancel" to proceed.

If the user clicks "OK", the box returns true. If the user clicks "Cancel", the box returns false.

Syntax:

```
window.confirm("sometext");
```

Figure 139 – The window.alert() method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dialog-boxes.php>)

The window.confirm() method can be written without the window prefix.

```
if (confirm("Press a button!")) {  
    txt = "You pressed OK!";  
} else {  
    txt = "You pressed Cancel!";  
}
```

Figure 140 – The window.confirm() method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dialog-boxes.php>)

Prompt Box

A prompt box is often used if you want the user to input a value before entering a page. When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed after entering an input value.

If the user clicks "OK" the box returns the input value. If the user clicks "Cancel" the box returns null.

Syntax:

```
window.prompt("sometext", "defaultText");
```

Figure 141 – The prompt box syntax (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dialog-boxes.php>)

The `window.prompt()` method can be written without the `window` prefix.

```
let person = prompt("Please enter your name", "Harry Potter");
let text;
if (person == null || person == "") {
  text = "User cancelled the prompt.";
} else {
  text = "Hello " + person + "! How are you today?";
}
```

Figure 142 – The `window.prompt()` method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dialog-boxes.php>)

Line Breaks

To display line breaks inside a popup box, use a back-slash followed by the character `n`.

```
alert("Hello\nHow are you?");
```

Figure 143 – Displaying line breaks inside a popup box (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-dialog-boxes.php>)

JavaScript Timing Events

Timing Events

The `window` object allows execution of code at specified time intervals.

These time intervals are called timing events.

The two key methods to use with JavaScript are:

- `setTimeout(function,milliseconds)`
Executes a function, after waiting a specified number of milliseconds.

- `setInterval(function,milliseconds)`
Same as `setTimeout()`, but repeats the execution of the function continuously.

The `setTimeout()` and `setInterval()` are both methods of the HTML DOM Window object.

The `setTimeout()` Method

```
window.setTimeout(function, milliseconds);
```

Figure 144 – The `setTimeout()` method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-timers.php>)

The `window.setTimeout()` method can be written without the window prefix.
The first parameter is a function to be executed.

The second parameter indicates the number of milliseconds before execution.

```
Example
Click a button. Wait 3 seconds, and the page will alert "Hello":

<button onclick="setTimeout(myFunction, 3000)">Try it</button>

<script>
function myFunction() {
  alert('Hello');
}
</script>
```

Figure 145 – The `window.setTimeout()` method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-timers.php>)

How to Stop the Execution?

The `clearTimeout()` method stops the execution of the function specified in `setTimeout()`.

```
window.clearTimeout(timeoutVariable)
```

Figure 146 – The `clearTimeout()` method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-timers.php>)

The `window.clearTimeout()` method can be written without the `window` prefix.

The `clearTimeout()` method uses the variable returned from `setTimeout()`:

```
myVar = setTimeout(function, milliseconds);  
clearTimeout(myVar);
```

Figure 147 – The `window.clearTimeout()` method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-timers.php>)

If the function has not already been executed, you can stop the execution by calling the `clearTimeout()` method:

```
Example  
Same example as above, but with an added "Stop" button:  
  
<button onclick="myVar = setTimeout(myFunction, 3000)">Try it</button>  
<button onclick="clearTimeout(myVar)">Stop it</button>
```

Figure 148 – The `clearTimeout()` method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-timers.php>)

The `setInterval()` Method

The `setInterval()` method repeats a given function at every given time-interval.


```
window.setInterval(function, milliseconds);
```

Figure 149 – The `setInterval()` method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-timers.php>)

The `window.setInterval()` method can be written without the `window` prefix.

The first parameter is the function to be executed.

The second parameter indicates the length of the time-interval between each execution.

This example executes a function called "myTimer" once every second (like a digital watch).

Example

Display the current time:

```
setInterval(myTimer, 1000);

function myTimer() {
  const d = new Date();
  document.getElementById("demo").innerHTML = d.toLocaleTimeString();
}
```

Figure 150 – The `window.setInterval()` method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-timers.php>)

How to Stop the Execution?

The `clearInterval()` method stops the executions of the function specified in the `setInterval()` method.

```
window.clearInterval(timerVariable)
```

Figure 151 – The `clearInterval()` method (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-timers.php>)

The `window.clearInterval()` method can be written without the `window` prefix.

The `clearInterval()` method uses the variable returned from `setInterval()`:

```
let myVar = setInterval(function, milliseconds);  
clearInterval(myVar);
```

Figure 152 – The variable returned from `setInterval()` (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-timers.php>)

Example

Same example as above, but we have added a "Stop time" button:

```
<p id="demo"></p>  
  
<button onclick="clearInterval(myVar)">Stop time</button>  
  
<script>  
let myVar = setInterval(myTimer, 1000);  
function myTimer() {  
  const d = new Date();  
  document.getElementById("demo").innerHTML = d.toLocaleTimeString();  
}  
</script>
```

Figure 153 – The variable returned from `setInterval()` (Source: <https://www.tutorialrepublic.com/javascript-tutorial/javascript-timers.php>)

5.4. JavaScript Advanced

JavaScript Date and Time

Note that the JavaScript Date object is provided by default as a global object in all JavaScript environments, thus you don't have to import it.

Date objects contain several methods for setting, getting, and manipulating dates and times.

Creating Date Objects

You can create a new Date object by passing the string representation of a date to the Date() constructor, or by passing the individual components of a date as parameters to the Date() constructor (year, month, day, hour, minute, second, millisecond).

The following examples create date objects:

- // Create a date object for today's date: `const now = new Date(); //`
- Outputs today's date and time in standard format `console.log(now); //`
- Outputs the day, date, month and year `console.log(now.toDateString()); //`
- Outputs the hour, minute and second `console.log(now.toTimeString());`
- Outputs the hour, minute, second and milliseconds `console.log(now.toLocaleString());`
- Outputs the year `console.log(now.getFullYear());`
- Outputs the month (from 0-11) `console.log(now.getMonth());`
- Outputs the day (from 1-31) `console.log(now.getDate());`
- Outputs the day of the week (from 0-6) `console.log(now.getDay());`
- Outputs the hour (from 0-23) `console.log(now.getHours());`
- Outputs the minute (from 0-59) `console.log(now.getMinutes());`
- Outputs the second (from 0-59) `console.log(now.getSeconds());`

- Outputs the millisecond (from 0-999) `console.log(now.getMilliseconds());`
- Outputs the number of milliseconds since the beginning of the day `console.log(now.getTime());`
- Outputs the number of seconds since 01.01.1970 `console.log(now.getTime() / 1000);`

Examples of outputs:

```
Thu, 22 Aug 2019 11:37:45 GMT
Thu Aug 22 2019 11:37:45 GMT+0000 (Coordinated Universal Time)
11:37:45 GMT+0000 (Coordinated Universal Time)
11:37:45 GMT+0000 (Coordinated Universal Time) 2019 8 22 4 11 37 45
Today is a Thursday, the 22nd of August, 2019.
```

You can check a practical example [here](#).

The `Date()` constructor can also take an integer value representing the number of milliseconds since 01.01.1970, or the string representation of a date.

If you pass an integer value representing the number of milliseconds since 01.01.1970, the constructor will create a date object representing this date and time.

If you pass a string representation of a date, the constructor will try to parse this date and create a date object representing this date and time. If the string cannot be parsed, the constructor will create an invalid date object.

Creating Date Objects With Moment.js

Moment.js is a JavaScript library which makes working with dates and times easy.

To use Moment.js in your project, you have to install it first with a package manager such as npm or yarn, as follows:

```
npm install moment yarn add moment
```

After the installation, you can import Moment.js into your JavaScript file with the import statement:

```
import moment from 'moment';
```

In the following example we use Moment.js to create a date object for today's date and time, and then we use the format() method to output the date and time in different formats:

```
import moment from 'moment'; // Create a date object for today's date  
and time const now = moment(); // Outputs today's date and time in  
standard format console.log(now.format()); // Outputs the day, date,  
month and year console.log(now.format('dddd, MMMM D, YYYY')); //  
Outputs the hour, minute and second console.log(now.format('h:mm:ss  
a')); // Outputs the day, date, month, year, hour, minute and // second  
console.log(now.format('ddd, hA'));
```

Outputs:

```
2019-08-22T15:25:33+04:00 Thursday, August 22, 2019 3:25:33 pm Thu, 3PM
```

Date Arithmetic

Date objects can be manipulated to change dates and times. Date objects have a set() method which can be used to set the year, month, day, hour, minute, second, and millisecond of a date.

When using set() , the object is updated in-place, thus you don't have to create a new Date object.

The following example uses the set() method to set the hour and minute of a date:


```
const now = new Date(); // Outputs the hour, minute and second
console.log(now.toLocaleString()); // Changes the hour to 12 and the
minute to 45 now.setHours(12); now.setMinutes(45);
console.log(now.toLocaleString());
```

Outputs:

```
11:56:09 AM 12:45:09 PM
```

The `setHours()` method can also be used to set the hour and minute, and the `setMinutes()` method can also be used to set the minute and second.

The `Date` object also has getters to get the year, month, day, hour, minute, second and millisecond of a date.

The following example uses the `get()` method to get the hour and minute of a date:

```
const now = new Date(); // Outputs the hour, minute and second
console.log(now.toLocaleString()); // Outputs the hour and minute
console.log(now.getHours() + ':' + now.getMinutes());
```

Outputs:

```
11:56:09 AM 11:56
```

When using the `set()` method, the object is updated in-place, thus you don't have to create a new `Date` object.

The `setDate()` method can be used to set the day of a date. The `setMonth()` method can be used to set the month of a date. The `setFullYear()` method can be used to set the year of a date.

The `getDate()` method can be used to get the day of a date. The `getMonth()` method can be used to get the month of a date. The `getFullYear()` can be used to get the year of a date.

The following example uses the `set()` , `setDate()` , `setMonth()` , `setFullYear()` , `get()` , `getDate()` , `getMonth()` and `getFullYear()` methods to set and get the year, month and day of a date:

```
const now = new Date(); now.setDate(1); // Sets the day to the first
day of the month now.setMonth(0); // Sets the month to January
now.setFullYear(2019); // Sets the year to 2019
console.log(now.toLocaleString()); now.setDate(now.getDate() + 10); //
Adds 10
```

JavaScript Math Operations

You can perform numerical operations using JavaScript. This can be done with the help of the `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division) and `%` (modulus) operators.

In the example below, we use the addition operator to add two numbers:

```
var x = 10; var y = 5; var z = x + y;
```

This will give the value of `z` as 15.

Similarly, we can use the - (subtraction), * (multiplication), / (division) and % (modulus) operators to perform their respective operations.

The modulus operator returns the remainder of a division operation. For example, if we divide 10 by 3, the remainder will be 1. Therefore, $10 \% 3$ will give the output 1.

Apart from these basic arithmetic operations, JavaScript also provides some built-in mathematical functions. These functions can be used to perform more complex operations.

Some of the most commonly used mathematical functions are:

- `Math.abs(x)` : This function returns the absolute value of a number. For example, `Math.abs(-10)` will return 10.
- `Math.ceil(x)` : This function rounds a number up to the nearest integer value. For example, `Math.ceil(4.7)` will return 5.
- `Math.floor(x)` : This function rounds a number down to the nearest integer value. For example, `Math.floor(4.7)` will return 4.
- `Math.max(x, y, ...)` : This function returns the maximum of the arguments passed to it. For example, `Math.max(10, 5, 20)` will return 20.
- `Math.min(x, y, ...)` : This function returns the minimum of the arguments passed to it. For example, `Math.min(10, 5, 20)` will return 5.
- `Math.pow(x, y)` : This function returns the value of x raised to the power of y. For example, `Math.pow(2, 3)` will return 8.
- `Math.random()` : This function returns a random number between 0 and 1.
- `Math.sqrt(x)` : This function returns the square root of x. For example, `Math.sqrt(16)` will return 4.

You can learn more about the Math object and its properties and methods from the JavaScript Math reference.

JavaScript Type Conversions

You can convert a value to a specific data type using built-in methods, such as `parseInt()` for converting a value to an integer, or `parseFloat()` for converting a value to a float.

You can also use the `Number()` method to convert a value to a number, or the `Boolean()` method to convert a value to a boolean.

In the following example we will convert a string to a number using the `Number()` method:

```
var x = "100"; var y = Number(x); console.log(y); // 100
```

In this example we have a string with the value "100". We convert this string to a number using the `Number()` method, and then we print the value of the number to the console.

You can also use the unary `+` operator to convert a value to a number. For example:

```
var x = "100"; var y = +x; console.log(y); // 100
```

In this example we have a string with the value "100". We use the unary `+` operator to convert the string to a number, and then we print the value of the number to the console.

You can use the `parseInt()` method to convert a string to an integer, or the `parseFloat()` method to convert a string to a float.

For example:

```
var x = "100.50"; var y = parseInt(x); console.log(y); // 100 var z =  
parseFloat(x); console.log(z); // 100.5
```

In this example we have a string with the value "100.50". We use the `parseFloat()` method to convert the string to an integer, and then we print the value of the integer to the console.

We also use the `parseFloat()` method to convert the string to a float, and then we print the value of the float to the console.

You can use the `Boolean()` method to convert a value to a boolean.

For example:

```
var x = "100"; var y = Boolean(x); console.log(y); // true
```

In this example we have a string with the value "100". We convert the string to a boolean using the `Boolean()` method, and then we print the value of the boolean to the console.

You can use the `!!` operator to convert a value to a boolean. For example:

```
var x = "100"; var y = !!x; console.log(y); // true
```

In this example we have a string with the value "100". We use the `!!` operator to convert the string to a boolean, and then we print the value of the boolean to the console.

If you want to convert a value to a string, you can use the `toString()` method.

For example:

```
var x = 100; var y = x.toString(); console.log(y); // "100"
```


In this example we have a number with the value 100. We convert the number to a string using the `toString()` method, and then we print the string to the console.

If you want to convert a value to an array, you can use the `Array.from()` method.

For example:

```
var x = "100"; var y = Array.from(x); console.log(y); // [ "1", "0", "0" ]
```

In this example we have a string with the value "100". We convert the string to an array using the `Array.from()` method, and then we print the array to the console.

If you want to convert a value to an object, you can use the `Object()` method.

For example:

```
var x = "100"; var y = Object(x); console.log(y); // { "0": "1", "1": "0", "2": "0" }
```

In this example we have a string with the value "100". We convert the string to an object using the `Object()` method, and then we print the object to the console.

JavaScript Event Listeners

DOM Event Listeners

JavaScript Event Listeners allow you to define functions that will run when a specific event occurs on an element in the DOM.

There are a number of different events that can occur on a DOM element, such as when the element is clicked, hovered over, or even when the element's contents are changed.

To add an event listener to an element, you first need to select the element using one of the DOM selection methods, such as `document.querySelector()` or `document.getElementById()`.

Once the element is selected, you can use the `addEventListener()` method to attach an event listener to the element.

The `addEventListener()` method takes two arguments: the name of the event to listen for, and a function to run when the event occurs.

For example, to add a click event listener to a button element, you would use the following code:

```
button . addEventListener ( "click" , function ( ) { console . log ( "The button was clicked!" ) ; } ) ;
```

In the code above, when the button element is clicked, the function passed to the `addEventListener()` method will be executed.

You can also pass a named function to the `addEventListener()` method, instead of an anonymous function:

```
function handleClick ( ) { console . log ( "The button was clicked!" ) ; } button . addEventListener ( "click" , handleClick ) ;
```

DOM Event Types

There are a number of different events that can occur on a DOM element.

The most common events are:

- click
- mouseover
- mouseout
- keypress
- change
- submit

You can find a complete list of DOM events on the Mozilla Developer Network.

DOM Event Object

When an event listener function is called, it is passed an event object as an argument.

The event object contains information about the event that occurred, such as the type of event, the element that the event occurred on, and any event-specific data.

For example, the click event object contains information about the click event, such as the x and y coordinates of the click.

For example, to log the x and y coordinates of a click event, you would use the following code:

```
button . addEventListener ( "click" , function ( ) { console . log ( "x: " + this . clientX + " , y: " + this . clientY ) ; } ) ;
```

In the code above, the `this` keyword refers to the event object, and the `clientX` and `clientY` properties are used to access the `x` and `y` coordinates of the click event.

DOM Event Delegation

DOM Event Delegation is a technique for attaching event listeners to elements that do not yet exist in the DOM.

This is useful when you have a large number of elements that you want to add event listeners to, but do not want to add an event listener to each element individually.

For example, if you have a list of 100 items, and you want to add a click event listener to each item, you could use event delegation to add a single click event listener to the parent element of the list, and have that listener function handle the click events for all of the child elements.

To use event delegation, you first need to select the parent element of the elements you want to add event listeners to.

You can then use the `addEventListener()` method to attach an event listener to the parent element.

The `addEventListener()` method takes two arguments: the name of the event to listen for, and a function to run when the event occurs.

The function passed to the `addEventListener()` method will be executed whenever the event occurs on any of the child elements of the parent element.

For example, to add a click event listener to all of the list items in a list, you would use the following code:

```
var listItems = document . querySelectorAll ( "li" ) ; listItems . forEach ( function ( listItem ) { listItem . addEventListener ( "click" , function ( ) { console . log ( "The list item was clicked!" ) ; } ) ; } ) ;
```

In the code above, a click event listener is added to each list item individually.

This can be inefficient if there are a large number of list items.

Instead, you could use event delegation to add a single click event listener to the parent element of the list items:

```
var list = document . querySelector ( "ul" ) ; list . addEventListener ( "click" , function ( event ) { if ( event . target . tagName === "LI" ) { console . log ( "The list item was clicked!" ) ; } } ) ;
```

In the code above, a click event listener is added to the parent element of the list items.

The function passed to the `addEventListener()` method checks the `tagName` property of the element that was clicked to see if it is an `` element.

If it is an `` element, then the function logs a message to the console.

This technique can be used to add event listeners to any type of element, not just list items.

For example, you could use event delegation to add a click event listener to all of the buttons in a document:


```
var buttons = document . querySelectorAll ( "button" ) ; buttons . forEach ( function ( button ) { button . addEventListener ( "click" , function ( ) { console . log ( "The button was clicked!" ) ; } ) ; } ) ;
```

Or you could use event delegation to add a click event listener to all of the links in a document:

```
var links = document . querySelectorAll ( "a" ) ; links . forEach ( function ( link ) { link . addEventListener ( "click" , function ( ) { console . log ( "The link was clicked!" ) ; } ) ; } ) ;
```

Event Delegation Summary

In this tutorial you learned about DOM event listeners in JavaScript.

You learned how to use the `addEventListener()` method to attach event listeners to elements in the DOM.

You also learned about the different events that can occur on a DOM element, and how to access the event object from within an event listener function.

Finally, you learned about DOM event delegation, and how to use it to attach event listeners to elements that do not yet exist in the DOM.

JavaScript Event Propagation

When an event occurs on an element, that event can be recognized by JavaScript and acted upon.

For example, if you click on a button on a web page, you can program JavaScript to recognize that event and take some action.

The event that occurred is a click event.

When you click on an element, the click event is fired on the element that was clicked.

The event then propagates up the DOM tree.

This means that if there is an event handler on a parent element, that event handler will be executed.

If there is an event handler on a grandparent element, that event handler will be executed.

This propagation up the DOM tree continues until the event reaches the root element of the DOM tree or until the event is stopped.

When the event reaches the root element, the event has bubbled all the way up the DOM tree.

The event will then propagate back down the DOM tree in the same way, from the root element to the element that was clicked.

This is called event **bubbling**. [The following diagram](#) shows event bubbling.

You can handle events on any element in the DOM tree.

You can also stop the propagation of an event.

For example, you might have an event handler on a parent element, and you do not want that event handler to be executed when the event occurs on a child element.

In that case, you can stop the event from propagating up the DOM tree.

This is called event propagation.

The event does not reach the root element, and it does not propagate back down the DOM tree.

You can also stop the propagation of an event at any point in the DOM tree.

For example, you might have an event handler on a grandparent element, and you do not want that event handler to be executed when the event occurs on a grandchild element.

In that case, you can stop the event from propagating down the DOM tree. This is called event capture.

In summary, when an event occurs on an element, that event can be recognized by JavaScript and acted upon.

The event propagates up the DOM tree, and then back down the DOM tree.

You can handle events on any element in the DOM tree.

You can also stop the propagation of an event.

JavaScript Borrowing Methods

We will use an example of an object that contains a property we need to borrow.

```
var myObject = {  
  
  someProperty: "foo",  
  
};
```

We can borrow the someProperty property from myObject like this:

```
var myProperty = myObject.someProperty;
```

myProperty will now contain the value "foo".

We can also use the same syntax to borrow methods from objects.

For example, if we have an object with a method we want to borrow:

```
var myObject = {  
  
  someMethod: function() {  
  
    // do something  
  
  }  
  
};
```

We can borrow the someMethod method like this:

```
var myMethod = myObject.someMethod;
```

We can now call myMethod like a normal function.

```
myMethod();
```

JavaScript Hoisting

Hoisting is a behaviour that is unique to JavaScript. It is a concept that is often misunderstood. Many developers believe that JavaScript hoists variables and function declarations to the top of the scope.

This is not entirely accurate. What JavaScript actually does is move declarations to the top of the scope, but not initialization. This can lead to some unexpected behaviour.

Consider the following code:

```
var foo = 1; function bar() { if (!foo) { var foo = 10; } console.log(foo); } bar();
```

What do you think the output of this code will be?

If you guessed 10, you would be wrong. The output of this code is 1.

This is because the declaration of foo is hoisted to the top of the scope, but the initialization is not. When the if statement is executed, foo is undefined and is thus set to 10.

This can be a bit confusing, but it is important to understand how hoisting works in JavaScript. It can help you to avoid some common mistakes.

JavaScript Closures

JavaScript closure is an inner function that has access to the variables in the outer (enclosing) function's scope chain. The closure has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function's variables, and it has access to the global variables.

JavaScript closures are created when the inner function is made within the outer function. Closures are used extensively in JavaScript libraries such as jQuery.

Here is a simple example of a closure in JavaScript:

```
function outerFunction(x) {  
  var innerFunction = function(y) {  
    return x + y;  
  }  
  return innerFunction;  
}  
  
var add5 = outerFunction(5);  
var add10 = outerFunction(10);  
  
console.log(add5(2)); // 7  
console.log(add10(2)); // 12
```

In the example above, we have a function `outerFunction` which has a single parameter `x`. This function contains a function `innerFunction` which has a single parameter `y`. The `innerFunction` returns the sum of `x` and `y`.

When we call the `outerFunction` with a value, it returns the `innerFunction`. When we call the function returned by the `outerFunction` with a value, it returns the sum of the value we passed to the `outerFunction` and the value we passed to the `innerFunction`.

In the example above, we have two closures: `add5` and `add10`. When we call `add5(2)`, it returns 7 because 5 is passed to the `outerFunction` as the value of `x`, and 2 is passed to the `innerFunction` as the value of `y`. When we call `add10(2)`, it returns 12 because 10 is passed to the `outerFunction` as the value of `x`, and 2 is passed to the `innerFunction` as the value of `y`.

A closure is a function that makes use of variables defined in outer functions that have previously returned. In the following example, the inner function `plus()` is making use of the variable `num` that was defined in the outer function `returnNotification()`:

```
function returnNotification() {  
  var num = 42;  
  
  function plus() {  
    return num + 1;  
  }  
  
  return plus;  
}  
  
var result = returnNotification();
```

```
console.log(result()); // 43
```

In the example above, we have a function `returnNotification()`. This function has a local variable `num` and a function `plus()`. The `plus()` function returns the value of `num` plus 1. The `returnNotification()` function returns the `plus()` function.

We assign the return value of the `returnNotification()` function to the variable `result`. When we call `result()`, it calls the `plus()` function and returns the value of `num` plus 1.

A closure is a function that makes use of variables defined in outer functions that have previously returned. In the following example, the inner function `plus()` is making use of the variable `num` that was defined in the outer function `returnNotification()`:

```
function returnNotification() {  
  var num = 42;  
  
  function plus() {  
    return num + 1;  
  }  
  
  return plus;  
}
```

```
var result = returnNotification();
```

```
console.log(result()); // 43
```

In the example above, we have a function `returnNotification()`. This function has a local variable `num` and a function `plus()`. The `plus()` function returns the value of `num` plus 1. The `returnNotification()` function returns the `plus()` function.

We assign the return value of the `returnNotification()` function to the variable `result`. When we call `result()`, it calls the `plus()` function and returns the value of `num` plus 1.

Closures are often used in JavaScript libraries such as jQuery. In the following example, we are using the jQuery library to create a button. When the button is clicked, an alert box is displayed with the text "Hello world!":

```
$(function() {  
  var button = $('button');  
  button.click(function() {  
    alert('Hello world!');  
  });  
});
```

In the example above, we have a button element with an id of "button". We are using the jQuery library to select the button element. We are then using the `click()` method to register a function that will be executed when the button is clicked.

The function that is executed when the button is clicked is a closure. It has access to the variables in the outer function, which in this case is the button element. When the function is executed, it displays an alert box with the text "Hello world!".

JavaScript Strict Mode

What is Strict Mode in JavaScript?

The strict mode is a way to opt in to a restricted variant of JavaScript. Strict mode isn't just a subset: it intentionally has different semantics from normal code. Browsers aren't required to run JS code in strict mode and many don't.

Strict mode makes it easier to write "secure" JavaScript. Strict mode changes previously accepted "bad syntax" into real errors.

SyntaxError: Unexpected eval or arguments in strict mode.

Strict mode also solves some mistakes that silently failed in JavaScript, throwing exceptions.

- Deleting a variable or a function.
- Using a variable that isn't declared.
- Deleting an object's property.
- Attempting to change a read-only property.
- Adding a property to an object that is not extensible.
- Using an object that has been frozen or sealed.
- Writing to a read-only property.
- Attempting to change the DontDelete attribute on a property.
- Assigning a value to a read-only global variable.
- Incrementing or decrementing a read-only property.
- Attempting to change the DontDelete attribute on a function.
- Assigning a value to a read-only global function.
- Attempting to change the prototype of a function.

The strict mode also prevents, or throws errors, when "unsafe" actions are taken.

- Using a variable without declaring it.

- Creating a global variable (without using the var statement).
- Deleting a variable or a function.
- Deleting an object's property.
- Using an object that has been frozen or sealed.
- Attempting to change the DontDelete attribute on a property.
- Assigning a value to a read-only global function.
- Incrementing or decrementing a read-only property.
- Attempting to change the prototype of a function.

In order to execute your code in strict mode, you need to add the "use strict"; directive to the beginning of your code. The directive is not a string, but rather a literal expression that appears at the top of a file, script, or function body.

```
"use strict";
```

The directive can be placed at either the beginning of a script or at the beginning of a function.

Example 1: Placing the directive at the beginning of a script.

```
"use strict"; x = 3.14; // This will cause an error because x is not declared.
```

Example 2: Placing the directive at the beginning of a function.

```
function myFunction() { "use strict"; y = 3.14; // This will cause an error because y is not declared. }
```

If the strict mode is used inside a function, the strict mode will only be applied to the function. If the strict mode is used at the top level of a script, the strict mode will be applied to the whole script.

If the strict mode is used at the top level of a script, the strict mode will be applied to the whole script.

If the strict mode is used inside a function, the strict mode will only be applied to the function.

In strict mode, any reference to an undeclared variable will cause a reference error.

```
function myFunction() { "use strict"; x = 3.14; // This will cause an error because x is not declared. }
```

In strict mode, any assignment to a non-writable global variable, a non-writable element of an arguments object, or a non-writable property of this will cause an error.

JavaScript JSON Parsing

What is JSON?

JSON stands for JavaScript Object Notation. It's a lightweight data-interchange format that allows you to exchange and store the data in an organized, easy-to-access manner. The best thing about it is that its syntax resembles plain English so even if you don't know anything about programming or coding, there are still chances of understanding what this code means!

Parsing JSON Data in JavaScript

The following example shows how to parse a string and create an object from it. You should always prefer parsing data as shown below:

```
var jsonString = '{"name":"John","age":30,"city":"New York"}'; var obj =  
JSON.parse(jsonString); console.log('Name: ',obj['name'], ' Age:',obj['age'], ' City:' , obj  
['city'] ); // Outputs Name : John, age : 30 and city as New york
```

Parsing Nested JSON Data in JavaScript

The following example shows how to parse nested data from a string into an object using the `eval()` function (which is not recommended). You can also use the same method for parsing nested objects but you need to be careful while doing so because if there are any errors then it will throw exceptions which might crash your program!

Encoding Data as JSON in JavaScript

The following example shows how to encode data into a string using the `JSON.stringify()` function.

```
var obj = { name : 'John', age: 30, city:'New York' }; var jsonString = JSON.stringify(obj);  
console.log('JSON String is ',jsonString ); // Outputs the stringified object as a JSON
```

JavaScript Error Handling

JavaScript is a very flexible language, which means that there are many ways to write code. This flexibility can lead to errors in your program if you're not careful about how you structure it.

One of the most common mistakes developers make is assuming that all code will execute correctly. This assumption can cause problems because JavaScript is an interpreted language, meaning that each line of code is executed as it's read by the interpreter. If there's an error in your code, the interpreter will stop executing at that point and throw an error message.

This tutorial will show you how to deal with errors gracefully so that your program can continue running even if there are some problems with the input data or other parts of the code.

The first thing you need to do is identify where the potential errors might occur in your code. For example, if you're reading data from a file, there's a possibility that the file doesn't exist or that it's been corrupted. If you're working with user input, there's a chance that the users will enter invalid data. Once you've identified these potential sources of error, you can start writing code to deal with them gracefully.

One common way to deal with errors is to use try/catch blocks. These blocks allow you to "try" some code and "catch" any errors that occur while it's executing. The syntax for a try/catch block looks like this:

```
try { // Code to try goes here } catch (error) { // Code to handle errors goes here }
```

The code inside the try block will be executed first. If no errors occur, the code in the catch block will never be executed. However, if an error does occur, execution will jump directly to the catch block and any subsequent code in the try block will be skipped.

Let's look at an example of how this works. Suppose you have a function that reads data from a file and prints it to the console. However, there's a possibility that the file doesn't exist:

```
function readFile(filename) { try { // Code to read and print the contents of "filename" goes here } catch (error) { console.log("Error reading file: " + error); } }
```

In this example, the readFile function will try to read and print the contents of a file. If an error occurs while reading the file, it will be caught by the catch block and printed to the console.

It's important to note that only errors that occur inside the try block will be caught by the catch block. If there's an error in the catch block itself (for example, if you forget to close a curly brace), it won't be caught and your program will crash.

Another way to deal with errors is to use JavaScript's built-in Error object. The Error object can be used to create custom error messages which can then be passed into a throw statement:

```
throw new Error("This is a custom error message");
```

When this code is executed, an error will be thrown and execution of the current function will stop. Any subsequent code in the function will not be executed.

You can also use the Error object to deal with errors that occur inside asynchronous code. For example, suppose you have a function that makes an HTTP request and returns the response data:

```
function makeRequest(url) { return new Promise((resolve, reject) => { http.get(url, (response) => { // Code to handle the response goes here }); }); }
```

This function returns a promise, which means that the data returned by the HTTP request will be available at some point in the future. If an error occurs while making the request (for example, if the URL is invalid), it will be caught by JavaScript's built-in error handling and rejected with an error message.

This rejection will cause the promise to be returned with an error, which can then be handled by the code that called makeRequest :


```
makeRequest("http://www.example.com") .then((data) => { // Code to handle the data goes here }) .catch((error) => { console.log("Error making request: " + error); });
```

In this example, we've registered a callback function for the "then" event which will be called if the promise is resolved successfully. We've also registered a callback function for the "catch" event which will be called if an error occurs. This allows us to deal with errors gracefully and continue executing our code even if there are some problems with the input data or other parts of the code.

JavaScript Regular Expressions

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

A regular expression is simply a sequence of characters that defines a particular search pattern. For example, the following regex would match any string containing an uppercase letter A: `/A/`.

Regular expressions can be used to perform a wide variety of tasks, such as:

- Extracting information from a given string (e.g., an email address);
- Validating user input (e.g., ensuring that a password is strong enough);
- Searching for and replacing text in a given string (e.g., changing all occurrences of "a" to "b"),
- Formatting text (e.g., adding commas between words),
- Generating random strings (e.g., creating unique IDs or passwords).
- And much more!

In JavaScript, regular expressions are also objects. These objects contain a set of methods that can be used to perform various operations on strings, such as searching for patterns or replacing text.

Creating Regular Expressions in JavaScript

There are two ways to create regular expressions in JavaScript: using a regular expression literal or the `RegExp()` constructor function.

A regular expression literal is simply a string that contains the pattern you want to match, enclosed within two forward slashes (`/`). For example:

```
const regex = /abc/ ; console .log(regex); // => /abc/
```

The above code creates a new regular expression object containing the pattern `abc` and assigns it to the variable `regex` . The value of this variable can then be used like any other JavaScript object. In particular, we can use its methods for performing various operations on strings.

The `RegExp()` constructor function can also be used to create regular expressions. This function takes two arguments: the first is a string containing the pattern you want to match, and the second is an optional "flags" argument that specifies certain settings for how the regex should work. For example:

```
const regex = new RegExp ( 'abc' ); console .log(regex); // => /abc/ const flagsRegex =  
new RegExp ( '123' , 'i' ); console .log(flagsRegex); // => /123/i; i stands for case-  
insensitive matching
```

The above code creates two new regular expression objects. The first contains the pattern `abc` and is case-sensitive (the default setting). The second object also has a

pattern of 123 but its matching will be case-insensitive, thanks to the `i` flag that was passed as the second argument to `RegExp()` .

Note: In JavaScript, regexes created using literal notation are immutable; they cannot have their properties or methods changed after they've been defined. On the other hand, regexes created with `RegExp()` can be modified at any time because it's just another constructor function like `Date` , `Array` , etc.

JavaScript Form Validation

JavaScript is a client-side scripting language, which means that the script runs on your visitor's computer. This allows you to do things like check if an email address has been entered correctly before it gets sent off to the server. It also reduces strain on your server because form data doesn't have to be submitted until after it has been checked for errors by JavaScript.

The first thing you need to do is create a form. You can use the following code:

```
<form action="yourpage.php" method="post"> <p>Name:</p><input type="text" name="name"><br /> <p>Email Address:</p><input type="text" name="email"><br /> </form>
```

This will create a basic form with two fields, one for the name and one for an email address. The `action` attribute tells the browser where to send the data when it is submitted, in this case `yourpage.php` . You can change this to whatever page you want to process the form data on. The `method` attribute specifies how the data should be sent, either `GET` or `POST`. In most cases you will want to use `POST` because it is more secure than `GET`, but there are some situations where using `GET` may be preferable (for example if you wanted people to be able bookmarked search results). For more

information about these methods see our [HTML Forms Tutorials: Introduction](#) article which covers them in detail.

The next thing we need do is add a submit button so that visitors can actually send us their details:

```
<input type="submit" value="Submit">
```

You can put this button anywhere inside the form tags. Now if you save your page and try it out, when you click on the submit button a new window will open with just `yourpage.php` in it (assuming that is what you set as the action). This happens because we haven't told our browser to do anything else yet - at present all submitting the form does is send us to another page which isn't very useful!

We need JavaScript to check for errors before sending off any data so let's add some code:

```
<script type="text/javascript"> function checkForm() { var errorMsg = ""; // Check each field to make sure it has a value. if (document.form1.name == "") { errorMsg += "- Please enter your name"; } else if (document.form1 .email == "" || document .form 1 .email_confirm != document form 1 email) {errorMsg += "- You must provide an e-mail address and confirm it by entering the same address again."; } return true; // This line submits the form only when there are no errors in any of the fields, otherwise we display our message telling them what they need to do: alert(errormsg); return false ; </script> <body onload="checkForm();" >
```

... rest of page here...

</body>

JavaScript Cookies

Creating a cookie in JavaScript is very simple. You just have to use the document object and its method `createCookie()`. The syntax of this function looks like:

```
document.cookie = "name=value; expires=date";
```

The name parameter represents the name of your cookie, while value stands for its content (string). If you want to set an expiration date on it, then add another attribute called expires with a valid `Date()` string as value or simply pass null if you don't need one.

```
document.cookie="username=John Doe;expires"+Date(30); //creates username cookies that will expire after 30 days from now
```

```
document.cookie="username=John Doe;expires"+Date(30*24); //creates username cookies that will expire after 30 days from now
```

Reading a cookie in JavaScript is very simple as well, you just have to use the document object and its method `getCookie()`. The syntax of this function looks like:

```
var name = getCookie("name");
```

The only parameter represents the name of your cookie (string). This function returns null if no such cookie exists or it has already expired. You can also read all existing cookies using another built-in property called `document.cookies` which contains an array with all available ones on your page separated by semicolons ";" . For example:


```
alert(document.cookies) ;//will display something like :
userName1=value1;userName2=value2 etc... /*this code displays nothing because
there are no any active cookies*/
```

Updating a cookie in JavaScript is very simple as well. You just have to create it again with the same name and value, but this time you also need to set an expiration date that's greater than its current one (if any). For example:

```
document.cookie = "name=newValue; expires=" + Date(Date().getTime()+1); /*this
code will update your cookies*/
document.cookie = "username=John Doe;expires"+Date(30*24); //creates username
cookies that will expire after 30 days from now /*this code creates new cookies if there
are no active ones or updates existing ones */
```

JavaScript Ajax Requests

Ajax is a web development technique for creating interactive web applications. The goal of Ajax is to make web pages feel more responsive by exchanging small amounts of data with the server behind the scenes, so that the entire page does not have to be reloaded each time the user makes a change.

In order to use Ajax, you will need to use a JavaScript library such as jQuery or Prototype. These libraries provide an easy way to send and receive data from a server without having to refresh the page.

Once you have included the Ajax library in your web page, you can start making requests to the server. For example, if you wanted to load some data from a file on the server, you would use the following code:

```
$.ajax({ url: 'data.json', success: function(data) { // do something with the data } });
```

This code will make an Ajax request to the URL 'data.json'. If the request is successful, the function in the 'success' callback will be executed with the data from the server as its argument.

If you want to send data to the server, you can use the '\$.ajax()' method's 'data' option:

```
$.ajax({ url: 'saveData.php', type: 'POST', data: { name: 'John', age: 20 } });
```

This code will make a POST request to the URL 'saveData.php'. The data that is being sent to the server is specified in the 'data' option as an object. In this example, we are sending two pieces of data: 'name' and 'age'.

The following example will show you how to make an Ajax GET request in JavaScript:

```
$.ajax({ url: 'getData.php', type: 'GET', success: function(data) { // do something with the data } });
```