JAVASCRIPT

References: **W3School**

Nguyen Quang Phu - Le Dinh Nguyen

1. Introduction

JavaScript can *display* data in different ways:

- Writing into an **HTML element**, using `innerHTML`.
- Writing into the **HTML output** using `document.write()` => FOR TESTING PURPOSE => USE `this` AFTER HTML DOCUMENTS IS LOADED, WILL DELETE ALL EXISITNG HTML
- Writing into an **alert box**, using `window.alert()`. => DISPLAY AN ALERT FROM THE PAGE
- Writing into the **browser console**, using `console.log()`. => AFTER RUN THE PAGE ON CHROME, PUSH F12 TO OPEN THE CONSOLE TO DEBUG

2. Variables

1. Always **declare** variables
2. Always use `const` if the value should not be changed
3. Always use `const` if the type should not be changed (Arrays and Objects)
4. Only use `let` if you can't use `const`
5. Only use `var` if you **MUST** support old browsers.
6. You cannot **re-declare** a variable declared with `let` or `const`, u can only do so with `var`
7. Variables defined with `let` can not be **redeclared**.
8. Variables defined with `let` must be **declared** before use.
9. Variables defined with `let` have **block scope**.
10. Variables defined with `const` cannot be **redeclared**.
11. Variables defined with `const` cannot be **reassigned**.
12. Variables defined with `const` have **block Scope**.
13. Use `const` when you declare:
    - A new Array
    - A new Object
    - A new Function
    - A new RegExp
14. The keyword `constant` defines a **constant** reference to a value

3. Data Types

3.1. Data Types

1. String
2. Number
3. Bigint
4. Boolean
5. Undefined
6. Null
7. Symbol
8. Object

3.2. Object Data Types

1. An object

2. An array
3. A date

> 📝**Note**:
>
> - You can use the JavaScript `typeof` operator to find the **type** of a JavaScript variable.
> - The `typeof` operator returns the **type** of a **variable** or an **expression**.

4. JavaScript Syntax

4.1. JavaScript Values

The JavaScript syntax defines two types of values:

- Fixed values, called Literals
- Variable values, called Variables

4.2. JavaScript Literals

The two most important syntax rules for fixed values are:

1. Numbers are written with or without decimals:

```
10.50
1000
```

2. Strings are text, written within double or single quotes:

```
"John Doe"
'John Doe'
```

4.3. JavaScript Operators

JavaScript uses **arithmetic operators** (`+ - * /`) to **compute** values:

Example

```
(6 + 5) / 10
```

JavaScript uses an **assignment operator** (=) to *assign* values to variables:

Example

```
let x, y;
x = 5;
y = 6;
```

5. JavaScript Events

> HTML events are "things" that happen to HTML elements. When JavaScript is used in HTML pages, JavaScript can "react" on these events.

5.1. HTML Events

An HTML event can be *something the browser does*, or *something a user does*. **Example** An HTML web page has finished loading An HTML input field was changed An HTML button was clicked

Often, when events happen, you may want to do something. JavaScript lets you **execute** code when events are **detected**. HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements.

Example

```
<element event = "some JavaScript">
```

In the following example, an `onclick` attribute (with code), is added to a `<button>` element:

```
<button onclick="document.getElementById('demo').innerHTML = Date()">The time is?
</button>
```

> **Note**: JavaScript code is often several lines long. It is more common to see event attributes calling functions

```
<button onclick="displayDate()">The time is?</button>

<script>
function displayDate() {
  document.getElementById("demo").innerHTML = Date();
}
</script>
```

5.2. Common HTML Events

Here is a **list of some common HTML events**:

Event	Description
`onchange`	An HTML element has been changed
`onclick`	The user clicks an HTML element
`onmouseover`	The user moves the mouse over an HTML element

Event	Description
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

The list is much longer: W3Schools JavaScript Reference HTML DOM Events

6. JavaScipt Strings

JavaScript strings are for **storing** and **manipulating** text.

6.1. String length

```
let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
let length = text.length;
```

6.2. Escape Character

The string will be **chopped** to "We are the so-called ". The **solution** to avoid this problem, is to use the **backslash escape character**.

Example

```
let text = "We are the so-called \"Vikings\" from the north.";
```

Code	Result
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Horizontal Tabulator
\v	Vertical Tabulator

6.3. Breaking Long Code Lines

> **Note**: The \ method is not the preferred method. It might not have universal support. Some browsers do not allow spaces behind the \ character.

6.4. String Methods

> **Note**: All string method return a new string, not modify the original one. String are immutable, cannot be changed, only replaced

`string_name.length`

Extracting string parts (count position from 0)

```
string_name.slice(start, end)
string_name.substring(start, end)
string_name.substr(start, end)
```

Replacing a string

`string_name.replace()`

- Not change the string it is called on, it returns a new string, replace only the first match
- By default, it is case sensitive, to ignore it, use `/i` flag (**insensitive**):

```
let newText = text.replace(/MIRCROSOFT/i, "QuangPhuDepZai");
```

- To replace all matches, use a `/g` flag (**global match**)

```
let newText = text.replace(/Microsoft/g, "W3Schools");
```

`string_name.replaceAll()`

`string_name.toUpperCase()` `string_name.toLowerCase()`

`concat()`

```
let text1 = "Hello";
let text2 = "World";
let text3 = text1.concat(" ", text2);
```

- String `trim()` -> remove space on **both sides** of a string
- String `trimStart()` -> remove **only white spaces from the beginning**
- String `trimEnd()` -> **only from the end** of the string
- String `padStart()` -> pads a string with another string until it reaches a given length
- String `padEnd()`

Extracting string

- String `charAt()`
- String `charCodeAt()`

```
Using []
let text = "HELLO WORLD";
let char = text[0];
```

- String `split()`

6.5. String Search

- String `indexOf()` -> first index of a found_str
- String `lastIndexOf()` -> last index of a found_str (both these return -1 if not found)
- String `search()` -> look nearly same as `indexOf`, but it can take a 2nd start pos argument
- String `match()` -> return an array containing the results of matching a `str` against a `str`
- String `matchAll()` -> return an iterator containing the results of matching a `str` agaisn a `str`
- String `includes()` -> return `true` if a str contains a specified value, otherwise `false`
- String `startsWith()` -> return `true` if a str begins with a specified value
- String `endsWith()` -> return `true` if a str ends with a specified value

6.6. String Template

Template Literal

Provide an easy way to **interpolate** variables and expressions into strings.

Variable substitutions

```
let firstName = "John";
let lastName = "Doe";
let text = `Welcome ${firstName}, ${lastName}!`;
```

Expression substitutions

```
let price = 10;
let VAT = 0.25;
let total = `Total: ${(price * (1 + VAT)).toFixed(2)}`;
```

HTML Templates

```
let header = "Templates Literals";
let tags = ["template literals", "javascript", "es6"];
```

```
let html = `<h2>${header}</h2><ul>`;
for (const x of tags) {
  html += `<li>${x}</li>`;
}

html += `</ul>`;
```

7. JavaScript Date

7.1. JavaScript Date Output

Example

```
const today = new Date(); //today
const d = new Date("2022-03-25");
```

> **Note**: Date objects are static. The "clock" is not "running". The computer clock is ticking, date objects are not.

7.2. Creating Date Objects

Date objects are created with the `new Date()` constructor.

There are **9 ways** to create a new date object:

```
new Date()
new Date(date string)

new Date(year,month)
new Date(year,month,day)
new Date(year,month,day,hours)
new Date(year,month,day,hours,minutes)
new Date(year,month,day,hours,minutes,seconds)
new Date(year,month,day,hours,minutes,seconds,ms)

new Date(milliseconds)
```

> **Note**: JavaScript counts months from 0 to 11 January = 0 December = 11

7.3. JavaScript Data Format

There are generally **3 types** of JavaScript **date input formats**:

Type	Example
ISO Date	"2015-03-25" (The International Standard)

Type	Example
Short Date	"03/25/2015"
Long Date	"Mar 25 2015" or "25 Mar 2015"

7.4. Get Date Methods

Method	Description
`getFullYear()`	Get year as a four digit number (yyyy)
`getMonth()`	Get month as a number (0-11)
`getDate()`	Get day as a number (1-31)
`getDay()`	Get weekday as a number (0-6)
`getHours()`	Get hour (0-23)
`getMinutes()`	Get minute (0-59)
`getSeconds()`	Get second (0-59)
`getMilliseconds()`	Get millisecond (0-999)
`getTime()`	Get time (milliseconds since January 1, 1970)

> **Note 1**: The get methods above return **Local time**.

> **Note 2**: The get methods return information from **existing date objects**. In a date object, the time is **static**. The "clock" is not "running". The time in a date object is **NOT** the same as current time.

7.5. Set Date Methods

Method	Description
`setDate()`	Set the day as a number (1-31)
`setFullYear()`	Set the year (optionally month and day)
`setHours()`	Set the hour (0-23)
`setMilliseconds()`	Set the milliseconds (0-999)
`setMinutes()`	Set the minutes (0-59)
`setMonth()`	Set the month (0-11)
`setSeconds()`	Set the seconds (0-59)
`setTime()`	Set the time (milliseconds since January 1, 1970)

8. JavaScript Comparisons

8.1. Comparison Operators

Operator	Description
==	equal to
===	equal to value and type
!=	not equal
!==	not equal to value and type
>	greater than
<	less than
>=	greater than and equal
>=	less than and equal

8.2. Logical Operations

Operator	Description
&&	and
||	or
!	not

8.3. Conditional (Ternary) Operator

Syntax

```
variablename = (condition) ? value_if_true : value_if_false;
```

9. JavaScript Loops

Different Kinds of Loops

- `for` - loops through a block of code a number of times
- `for/in` - **loops through the properties** of an **object**
- `for/of` - **loops through the values** of an **iterable object**
- `while` - loops through a block of code while a specified condition is `true`
- `do/while` - also loops through a block of code while a specified condition is `true`

9.1. For loops

```
for (let i = 0; i < 5; i++) {
  text += "The number is " + i + "<br>";
}
```

```
for (let i = 0, len = cars.length, text = ""; i < len; i++) {
  text += cars[i] + "<br>";
}
```

9.2. For In loop

The JavaScript `for in` statement loops **through the properties** of an Object:

```
const person = {fname:"John", lname:"Doe", age:25};

let text = "";
for (let x in person) {
  text += person[x];
}
```

9.3. For Of loop

The JavaScript for of statement loops **through the values** of an **iterable** object, it lets you loop over **iterable data structures** such as **Arrays, Strings, Maps, NodeLists**, and more:

Looping over an Array

```
const cars = ["BMW", "Volvo", "Mini"];

let text = "";
for (let x of cars) {
  text += x;
}
```

Looping over an String

```
let language = "JavaScript";

let text = "";
for (let x of language) {
text += x;
}
```

9.4. While loop

The `while` loop loops through a **block of code** as long as a specified condition is true.

```
while (i < 10) {
  text += "The number is " + i;
```

```
  i++;
}
```

9.5. Do While Loop

The `do while` loop is a **variant** of the while loop. This loop will **execute the code block once, before checking if the condition is true**, then it will **repeat** the loop as long as the condition is `true`.

```
do {
  text += "The number is " + i;
  i++;
}
while (i < 10);
```

10. Type of

In JavaScript there are **5 different data types** that can contain values:

- `string`
- `number`
- `boolean`
- `object`
- `function`

There are **6 types of objects**:

- `Object`
- `Date`
- `Array`
- `String`
- `Number`
- `Boolean`

And 2 data types that **cannot contain values** `null` = `undefined`

10.1. The typeof Operator

You can use the `typeof` operator to **find the data type** of a JavaScript variable.

Example

```
typeof "John"                 // Returns "string"
typeof 3.14                   // Returns "number"
typeof NaN                    // Returns "number"
typeof false                  // Returns "boolean"
typeof [1,2,3,4]              // Returns "object"
typeof {name:'John', age:34}  // Returns "object"
typeof new Date()             // Returns "object"
```

```
typeof function () {}         // Returns "function"
typeof myCar                  // Returns "undefined" *
typeof null                   // Returns "object"
```

> **Note**:
>
> - The data type of NaN is number
> - The data type of an array is object
> - The data type of a date is object
> - The data type of null is object
> - The data type of an undefined variable is **undefined** *
> - The data type of a variable that has not been assigned a value is also **undefined** *

10.2. Primitive Data

A **primitive data** value is a single simple data value with *no additional properties and methods*.

The `typeof` operator can return one of these **primitive types**:

- `string`
- `number`
- `boolean`
- `undefined`

10.3. Complex Data

- The `typeof` operator can return one of two complex types:
  - `function`
  - `object`
- The `typeof` operator returns "object" for **objects**, **arrays**, and **null**.
- The `typeof` operator does not return "object" for **functions**

```
typeof {name:'John', age:34} // Returns "object"
typeof [1,2,3,4]             // Returns "object" (not "array", see note below)
typeof null                  // Returns "object"
typeof function myFunc(){}   // Returns "function"
```

10.4. The Data Type of typeof

The `typeof` operator is **not a variable**. It is an **operator**. Operators (`+ - * /`) do **not have any data type**.

But, the `typeof` operator always **returns a string** (containing the **type of the operand**).

10.5. The constructor Property

The `constructor` property returns the **constructor function** for all JavaScript variables.

```
"John".constructor                 // Returns function String()  {[native code]}
(3.14).constructor                 // Returns function Number()  {[native code]}
false.constructor                  // Returns function Boolean() {[native code]}
[1,2,3,4].constructor              // Returns function Array()   {[native code]}
{name:'John',age:34}.constructor   // Returns function Object()  {[native code]}
new Date().constructor             // Returns function Date()    {[native code]}
function () {}.constructor         // Returns function Function(){[native code]}
```

10.6. Undefined

A variable without a value, has the value `undefined` and the type is also `undefined`.

```
let car;    // Value is undefined, type is undefined
```

10.7. Empty Values

- An **empty value** has nothing to do with `undefined`
- An **empty string** has both a **legal value** and a **type**.

```
let car = "";    // The value is "", the typeof is "string"
```

10.8. Null

In JavaScript `null` is "nothing". It is supposed to be something that **doesn't exist**.

> **Note**: The data type of `null` is an object.

```
let person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = null;    // Now value is null, but type is still an object
```

10.9. The instanceof Operator

The `instanceof` operator returns `true` if an object is an instance of the specified object:

```
const cars = ["Saab", "Volvo", "BMW"];

(cars instanceof Array);
(cars instanceof Object);
(cars instanceof String);
(cars instanceof Number);
```

10.10. The void Operator

- The **void** operator evaluates an expression and returns **undefined**. - This operator is often used to **obtain the undefined primitive value**, using "void(0)" (useful when evaluating an expression without using the return value).

```
<a href="javascript:void(0);">
  Useless link
</a>

<a href="javascript:void(document.body.style.backgroundColor='red');">
  Click me to change the background color of body to red
</a>
```

11. Type conversion

11.1. JavaScript Type Conversion

JavaScript variables can be converted to a new variable and another data type:

- By the use of a JavaScript function
- **Automatically** by JavaScript itself

11.2. Converting Strings to Numbers

The global method `Number()` converts a variable (or a value) into a number.

> An empty string (like "") converts to 0. A non numeric string (like "John") converts to `NaN` (Not a Number).

11.3. Number Methods

Methods that can be used to convert strings to numbers:

Method	Description
`Number()`	Returns a number, converted from its argument
`parseFloat()`	Parses a string and returns a floating point number
`parseInt()`	Parses a string and returns an integer

11.4. The Unary + Operator

The **unary + operator** can be used to convert a variable to a number:

```
let y = "5";      // y is a string
let x = + y;      // x is a number
```

If the variable cannot be converted, it will still become a number, but with the value NaN (Not a Number):

```
let y = "John";   // y is a string
let x = + y;      // x is a number (NaN)
```

11.5. Converting Numbers to Strings

- The global method `String()` can convert numbers to strings.
- It can be used on any type of numbers, literals, variables, or expressions:

```
String(x)         // returns a string from a number variable x
String(123)       // returns a string from a number literal 123
String(100 + 23)  // returns a string from a number from an expression
```

12. Bitwise

12.1. JavaScript Bitwise Operators

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
!	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shifts left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shifts right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off
>>>	Zero fill right shift	Shifts right by pushing zeros in from the left, and let the rightmost bits fall off

13. RegExp

13.1. What Is a Regular Expression?

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

13.2. Syntax

```
/pattern/modifiers;
```

Using **String Methods** In JavaScript, regular expressions are often used with the two **string methods**: `search()` and `replace()`

- The `search()` method uses an expression to search for a match, and returns the position of the match.
- The `replace()` method returns a modified string where the pattern is replaced.

Using String `search()` With a String

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

```
let text = "Visit W3Schools!";
let n = text.search("W3Schools"); // 6
```

Regular Expression Modifiers

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

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

Regular Expression Patterns

Brackets are used to find a range of characters:

Expression	Description
`[abc]`	Find any of the characters between the brackets
`[0-9]`	Find any of the digits between the brackets
`(x!y)`	Find any of the alternatives separated with

14. Precedence

Operator precedence describes the order in which operations are performed in an arithmetic expression. Multiplication (*) and division (/) have **higher precedence** than addition (+) and subtraction (-).

15. Errors

15.1. Throw, and Try...Catch...Finally

The `try` statement defines a code block to run (to try). The `catch` statement defines a code block to **handle** any error. The `finally` statement defines a code block to **run regardless of the result**. The `throw` statement defines a custom error.

Example

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

<script>
try {
  adddlert("Welcome guest!");
}
catch(err) {
  document.getElementById("demo").innerHTML = err.message;
}
</script>
```

> **Note**: JavaScript catches **alert as an error**, and **executes the catch code** to handle it.

15.2. JavaScript try and catch

- The `try` statement allows you to define a block of code to be **tested for errors** while it is being executed.
- The `catch` statement allows you to define a block of code to be **executed, if an error occurs** in the try block.
- The JavaScript statements try and catch come in pairs:

```
try {
  // Block of code to try
}
catch(err) {
  // Block of code to handle errors
}
```

15.3. JavaScript Throws Errors

When an error occurs, JavaScript will normally **stop** and generate an **error message**.

The technical term for this is: JavaScript will **throw an exception (throw an error)**

> **Note**: JavaScript will actually create an **Error object** with two properties: **name** and **message**.

15.4. The throw Statement

- The `throw` statement allows you to **create a custom error**.
- Technically you can **throw an exception (throw an error)**

- The **exception** can be a JavaScript `String`, a `Number`, a `Boolean` or an `Object`:

```
throw "Too big";    // throw a text
throw 500;          // throw a number
```

If you use `throw` together with `try` and `catch`, you can **control** program flow and **generate** custom error messages.

15.5. Input Validation Example

This example examines input. If the *value is wrong, an exception (`err`) is thrown*.

The exception (`err`) is **caught by the catch statement** and a custom error message is displayed:

```
<p>Please input a number between 5 and 10:</p>

<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>
<p id="p01"></p>

<script>
function myFunction() {
  const message = document.getElementById("p01");
  message.innerHTML = "";
  let x = document.getElementById("demo").value;
  try {
    if(x.trim() == "") throw "empty";
    if(isNaN(x)) throw "not a number";
    x = Number(x);
    if(x < 5) throw "too low";
    if(x > 10) throw "too high";
  }
  catch(err) {
    message.innerHTML = "Input is " + err;
  }
}
</script>
```

15.6. HTML Validation

Modern browsers will often use a **combination of JavaScript and built-in HTML validation**, using predefined validation rules defined in HTML attributes:

```
<input id="demo" type="number" min="5" max="10" step="1">
```

15.7. The finally Statement

The **finally** statement lets you execute code, after try and catch, regardless of the result:

Syntax:

```
try {
  // Block of code to try
}
catch(err) {
  // Block of code to handle errors
}
finally {
  // Block of code to be executed regardless of the try / catch result
}
```

Example

```
function myFunction() {
  const message = document.getElementById("p01");
  message.innerHTML = "";
  let x = document.getElementById("demo").value;
  try {
    if(x.trim() == "") throw "is empty";
    if(isNaN(x)) throw "is not a number";
    x = Number(x);
    if(x > 10) throw "is too high";
    if(x < 5) throw "is too low";
  }
  catch(err) {
    message.innerHTML = "Error: " + err + ".";
  }
  finally {
    document.getElementById("demo").value = "";
  }
}
```

15.8. The Error Object

JavaScript has a built in error object that provides error information when an error occurs.

The error object provides **two useful properties**: **name** and **message**.

Error Object Properties

Property	Description
name	Sets or returns an error name
message	Sets or returns an error message (a string)

Error Name Values

Six different values can be returned by the error name property:

Error Name	Description
EvalError	An error has occurred in the `eval()` function
RangeError	A number "out of range" has occurred
ReferenceError	An illegal reference has occurred
SyntaxError	A syntax error has occurred
TypeError	A type error has occurred
URIError	An error in `encodeURI()` has occurred

The six different values are described below.

16. Scope

Scope determines the **accessibility (visibility)** of variables. JavaScript has **3 types** of scope:

- **Block** scope
- **Function** scope
- **Global** scope

16.1. Block Scope

- Before **ES6** (2015), JavaScript had only **Global Scope** and **Function Scope**
- **ES6** introduced two important new JavaScript keywords: `let` and `const`.
- These two keywords provide **Block Scope** in JavaScript.
- Variables declared inside a **{ } block** cannot be accessed from outside the block:

```
{
  let x = 2;
}
// x can NOT be used here
```

- Variables declared with the `var` keyword can **NOT** have block scope.
- Variables declared inside a **{ } block** can be accessed from **outside** the block.

```
{
  var x = 2;
}
// x CAN be used here
```

16.2. Local Scope

Variables declared within a JavaScript function, become **LOCAL** to the function.

```
// code here can NOT use carName
function myFunction() {
  let carName = "Volvo";
  // code here CAN use carName
}
// code here can NOT use carName
```

16.3. Global Scope

- Variables declared **Globally** (outside any function) have **Global Scope**.
- **Global** variables can be accessed from anywhere in a JavaScript program.
- Variables declared with `var`, `let` and `const` are quite similar when declared outside a block.

```
var x = 2;       // Global scope
let x = 2;       // Global scope
const x = 2;       // Global scope
```

16.4. Automatically Global

If you **assign a value to a variable** that has **not** been **declared**, it will automatically become a **GLOBAL** variable.

This code example will declare a global variable `carName`, even if the value is assigned inside a function.

```
myFunction();

// code here can use carName

function myFunction() {
  carName = "Volvo";
}
```

17. Hoisting

17.1. JavaScript Declarations are Hoisted

In JavaScript, a variable **can be declared after it has been used**.

```
x = 5; // Assign 5 to x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x;                    // Display x in the element
```

```
var x; // Declare x
```

Hoisting is JavaScript's **default behavior** of **moving all declarations to the top of the current scope**

17.2. The let and const Keywords

- Variables defined with `let` and `const` are **hoisted to the top** of the **block**, but **not initialized**.
- **Meaning**: The block of code is aware of the variable, but it cannot be used until it has been declared.
- Using a `let` variable before it is declared will result in a `ReferenceError`.

17.3. JavaScript Initializations are Not Hoisted

> JavaScript only **hoists declarations, not initializations**.

Example 1

```
var x = 5; // Initialize x
var y = 7; // Initialize y

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y
```

Example 2

```
var x = 5; // Initialize x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y

var y = 7; // Initialize y
```

18. Strict Mode

Declaring Strict Mode

Strict mode is **declared by adding** `"use strict"`; to the beginning of a scope.

```
"use strict";
x = 3.14;       // This will cause an error because x is not declared
```

```
"use strict";
myFunction();
```

```
function myFunction() {
  y = 3.14;   // This will also cause an error because y is not declared
}
```

```
x = 3.14;       // This will not cause an error.
myFunction();

function myFunction() {
  "use strict";
  y = 3.14;   // This will cause an error
}
```

19. JavaScript Modules

19.1. Modules

- JavaScript modules allow you to **break up your code into separate file**, makes it easier to **maintain a code-base**.
- Modules are imported from **external files** with the `import` statement.
- Modules also rely on `type="module"` in the `<script>` tag.

```
<script type="module">
import message from "./message.js";
</script>
```

19.2. Export

Modules with functions or variables can be stored in any **external file**.

There are **two types of exports**: **Named Exports** and **Default Exports**.

19.3. Named Exports

You can create named exports **two ways**. **In-line** individually, or **all at once** at the bottom.

In-line individually

```
export const name = "Jesse";
export const age = 40;
```

All at once at the bottom

```
const name = "Jesse";
const age = 40;

export {name, age};
```

19.4. Default Exports

> You can only have one default export in a file.

Example

```
const message = () => {
const name = "Jesse";
const age = 40;
return name + ' is ' + age + 'years old.';
};

export default message;
```

19.5. Import

You can **import modules** into a file in **two ways**, based on if they are **named exports or default exports**.

> Named exports are constructed using curly braces. Default exports are not.

Import from named exports

Import named exports from the file person.js:

```
import { name, age } from "./person.js";
```

Import from default exports

Import a default export from the file message.js:

```
import message from "./message.js";
```

> **Note**: Modules only work with the HTTP(s) protocol. A web-page opened via the file:// protocol cannot use import export.

20. JSON

20.1. JSON Objects

JSON objects are written **inside curly braces**.

Just like in JavaScript, objects can **contain multiple name/value pairs**

```
{"firstName":"John", "lastName":"Doe"}
```

20.2. JSON Arrays

JSON arrays are written **inside square brackets**.

Just like in JavaScript, an **array can contain objects**:

```
"employees":[
  {"firstName":"John", "lastName":"Doe"},
  {"firstName":"Anna", "lastName":"Smith"},
  {"firstName":"Peter", "lastName":"Jones"}
]
```

20.3. Converting a JSON Text to a JavaScript Object

```
let text = '{ "employees" : [' +
'{ "firstName":"John" , "lastName":"Doe" },' +
'{ "firstName":"Anna" , "lastName":"Smith" },' +
'{ "firstName":"Peter" , "lastName":"Jones" } ]}';

const obj = JSON.parse(text);
```

21. A function

21.1. Function Definition

Function Declarations

This is an external link to genome.gov

```
function functionName(parameters) {
  // code to be executed
}
```

Function Expressions

A function expression can be **stored** in a variable:

```
const x = function (a, b) { return a*b };
```

After a function expression has been stored in a variable, the **variable can be used as a function**:

```
const x = function (a, b) { return a*b };
let z = x(4, 3);
```

Function Hoisting

- **Hoisting** is JavaScript's **default** behavior of *moving declarations* to the *top* of the *current scope*.
- **Hoisting** applies to *variable declarations* and to *function declarations*.
- Example:

```
myFunction(5);

function myFunction(y) {
  return y * y;
}
```

Self-Invoking Function

- A self-invoking expression is **invoked (started) automatically**, without being called.
- Function expressions will execute automatically if the expression is followed by `()`.
- You *cannot self-invoke a function declaration*.
- You have to **add parentheses around** the function to indicate that it is a function expression.
- **Example**:

```
(function () {
  let x = "Hello!!";  // I will invoke myself
})();
```

> **Note**: The function above is actually an **anonymous self-invoking function** (function without name).

Functions are Objects

JavaScript functions can best be described as objects. They have both **properties** and **methods**.

- `arguments.length` returns the number of arguments received.
- `toString()` returns the function as a string. **Example**:

```
function myFunction(a, b) {
return a * b;
}

let text = myFunction.toString();
// string return: function myFunction(a, b) { return a * b; }
```

Arrow Functions

A **short syntax** for writing function expressions, do not need the `function`, `return` keyword and the **curly brackets**.

> **Note:**
>
> - Arrow functions are **not hoisted**. They must be *defined before* they are *used*.
> - Using `const` is safer than using `var`, because a function expression is always **constant value**.

Example:

```
const x = (x, y) => { return x * y };
```

21.2. Function Parameters

Parameter Rules

- JavaScript function definitions do **not specify data types** for **parameters**.
- JavaScript functions do **not perform type checking** on the **passed arguments**.
- JavaScript functions do **not check** the **number** of **arguments received**.
- If a function is called with **missing arguments** (less than declared), the **missing values** are set to **undefined**.
- Default parameter values

Example:

```
function myFunction(x, y = 10) {
return x + y;
}
myFunction(5);
```

Function Rest Parameters

The rest parameter (`...`) allows a function to *treat an indefinite number of arguments as an array*.

Example:

```
function sum(...args) {
    let sum = 0;
    // Iterating all elements in arr args
    for (let arg of args) {
        sum += arg;
    }
    return sum;
}

let x = sum(4, 9, 16, 25, 29, 100, 66, 77);
```

The arguments Object

JavaScript functions have a **built-in object** called the `arguments` object.

The argument object contains an **array of the arguments** used when the function was called (invoked).

Example:

```
// Calculate the Sum of all input values
x = sumAll(1, 123, 500, 115, 44, 88);

function sumAll() {
  let sum = 0;
  for (let i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }
  return sum;
}
```

> **Note**:
>
> - If a function is called with too many **arguments (more than declared)**, these arguments can be reached using the arguments object.
> - Arguments are passed by Value: **Changes** to **arguments** are **not visible (reflected) outside** the function.
> - Objects are passed by reference: **Changes** to **object properties** are **visible (reflected) outside** the function.

21.3. Function Invocation

Invoking a JavaScript Function as Method

```
const myObject = {
  firstName:"John",
  lastName: "Doe",
  fullName: function () {
    return this.firstName + " " + this.lastName;
  }
}
myObject.fullName();         // Will return "John Doe"
```

`this` keyword

- In JavaScript, the `this` keyword refers to an **object**.
- In an **object** method, `this` refers to the **object**.
- Alone, `this` refers to the **global object**.
- In a **function**, `this` refers to the **global object**.
- In a **function**, in **strict mode**, `this` is `undefined`.
- In an **event**, `this` refers to the element that received the event.
- Methods like `call()`, `apply()`, and `bind()` can refer `this` to any object.
- `this` is **not a variable**. It is a **keyword**. You cannot change the value of `this`.

21.4. JavaScript function `call()`

The `call()` method is a **predefined** JavaScript method.

It can be used to invoke (call) a method with an **owner object as an argument** (parameter).

With `call()`, an object *can use a method belonging to another object*.

`call()` method

```
const person = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}
const person1 = {
  firstName:"John",
  lastName: "Doe"
}
const person2 = {
  firstName:"Mary",
  lastName: "Doe"
}

// This will return "John Doe":
person.fullName.call(person1);
```

`call()` method with arguments

```
const person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + "," + city + "," + country;
  }
}

const person1 = {
  firstName:"John",
  lastName: "Doe"
}

person.fullName.call(person1, "Oslo", "Norway");
```

21.5. JavaScript function `apply()`

With the `apply()` method, you can write a method that can be used on different objects

`apply()` method

Similar to `call()` method.

```
const person = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}

const person1 = {
  firstName: "Mary",
  lastName: "Doe"
}

// This will return "Mary Doe":
person.fullName.apply(person1);
```

Difference between `call()` and `apply()`

The difference is:

- The `call()` method takes arguments **separately**.
- The `apply()` method takes arguments as an **array**.

`apply()` method with arguments

Accept arguments in an array

```
const person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + "," + city + "," + country;
  }
}

const person1 = {
  firstName:"John",
  lastName: "Doe"
}

person.fullName.apply(person1, ["Oslo", "Norway"]);

// COMPARE WITH THE call() method
const person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + "," + city + "," + country;
  }
}

const person1 = {
  firstName:"John",
  lastName: "Doe"
}

person.fullName.call(person1, "Oslo", "Norway");
```

Max method on Arrays

You can find the **largest number** (in a list of numbers) using the `Math.max()` method

```
Math.max(1,2,3);  // Will return 3
Math.max.apply(null, [1,2,3]); // Will also return 3
```

21.6. JavaScript Function `bind()`

Function Borrowing

- With the `bind()` method, an object can **borrow a method from another** object.
- The example below creates 2 objects (person and member).
- The member object **borrows** the fullname method from the person object

```
const person = {
  firstName:"John",
  lastName: "Doe",
  fullName: function () {
    return this.firstName + " " + this.lastName;
```

```
  }
}

const member = {
  firstName:"Hege",
  lastName: "Nilsen",
}

let fullName = person.fullName.bind(member);
```

Preserving `this`

Sometimes `bind()` method has to be used to prevent losing `this`

In the following example, the **person object** has a **display method**. In the display method, `this` refers to the person object

```
const person = {
  firstName:"John",
  lastName: "Doe",
  display: function () {
    let x = document.getElementById("demo");
    x.innerHTML = this.firstName + " " + this.lastName;
  }
}

person.display();
```

BUT when a function is used as a **callback**, `this` is lost.

```
const person = {
  firstName:"John",
  lastName: "Doe",
  display: function () {
    let x = document.getElementById("demo");
    x.innerHTML = this.firstName + " " + this.lastName;
  }
}
// Display after 3 seconds
setTimeout(person.display, 3000); // output: undefined undefined
```

Use the `bind()` method to solve the problem

```
const person = {
  firstName:"John",
  lastName: "Doe",
  display: function () {
```

```
    let x = document.getElementById("demo");
    x.innerHTML = this.firstName + " " + this.lastName;
  }
}

// the bind() method is used to bind person.display to person
let display = person.display.bind(person);
setTimeout(display, 3000);
```

21.7. JavaScript Closures

Global variables can be made **local** (**private**) with **closures**.

Global Variables

- A `function` can access all variables defined **inside** the function and **outside** the function
- **Global** and **local** variables with the **same name** are **different variables**. Modifying one does not affect the other.
- Variables **created without a declaration keyword** (var, let, or const) are always **global**, even if they are created inside a function.

Variable Lifetime

- Global variables live **until** the page is **discarded**
- Local variables have **short lives**: **created** when the function is **invoked**, and **deleted** when the function is **finished**.

JavaScript Closures

```
// Self-Invoking function
// add becomes a function
const add = (function () {
  let counter = 0;
  return function () {counter += 1; return counter}
})();

add();
add();
add();

// the counter is now 3
```

22. JavaScript Objects

22.1. Objects Definition

Almost "everything" is an object.

- **Booleans** can be objects (if defined with the new keyword)
- **Numbers** can be objects (if defined with the new keyword)
- **Strings** can be objects (if defined with the new keyword)
- **Dates** are always objects
- **Maths** are always objects
- **Regular expressions** are always objects
- **Arrays** are always objects
- **Functions** are always objects
- **Objects** are always objects

All JavaScript values, except primitives, are objects.

Primitives

- A **primitive value** is a value that has **no properties** or **methods**.
- A **primitive data type** is data that has a primitive value. 7 types: string, number, boolean, null, undefined, symbol, bigint
- Primitive values are **immutable** (*hardcoded* and *cannot be changed*).

> **Note**:
>
> - Objects are **Variables**
> - Using **Object Literal** is the easiest way to create a JavaScript Object (can both define and create an object in 1 statement)
> - JavaScript Objects are **mutable**. They are **addressed by reference, not by value**.

Example

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};

// Create an empty object, then adds properties
const person1 = {};
person1.firstName = "John";
person1.lastName = "Doe";
person1.age = 50;
person1.eyeColor = "blue";

// Objects are Mutable
const x = person; // not a copy of a person, both x and person are the same
objects
x.age = 18 // change both property age of x and person
```

22.2. Objects Properties

- A JavaScript object is a collection of unordered properties.
- Accessing the property of an object:
  - `objectName.property`
  - `objectName["property"]`

Loop through properties of Objects and "do sth" with the object

```
const person = {
  fname:" John",
  lname:" Doe",
  age: 25
};

for (let x in person) {
  txt += person[x];
}

person.nationality = "English"; // add property
delete person.nationality; // delete a property
```

Nested Arrays and Objects

```
const myObj = {
  name: "John",
  age: 30,
  cars: [
    {name:"Ford", models:["Fiesta", "Focus", "Mustang"]},
    {name:"BMW", models:["320", "X3", "X5"]},
    {name:"Fiat", models:["500", "Panda"]}
  ]
}

// Access arrays inside arrays, use a for-in loop for each array
for (let i in myObj.cars) {
  x += "<h1>" + myObj.cars[i].name + "</h1>";
  for (let j in myObj.cars[i].models) {
    x += myObj.cars[i].models[j];
  }
}
```

22.3. Objects Methods

```
const person = {
  firstName: "John",
  lastName: "Doe",
```

```
  id: 5566,
  city: "New York
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
};

// Access the fullName() method
name = person.fullName();

// Access the fullName property, return the function definition
name = person.fullName;
```

22.4. Objects Display

Some common solutions are:

- Displaying the Object Properties by **name**
- Displaying the Object Properties in a **Loop**
- Displaying the Object using `Object.values()` (*Any object can be converted to an array using this function*)
- Displaying the Object using `JSON.stringify()` (*Any object can be stringified - or convert to a string - with the this function*)

```
const person = {
  name: "John",
  age: 30,
  city: "New York"
};

// 1st
let output = person.name + "," + person.age + "," + person.city;

// 2nd
let txt = "";
for (let x in person) {
    txt += person[x] + " ";
};

// 3rd
// myArray is now a JavaScript array
const myArray = Object.values(person);

// 4th
let myString = JSON.stringify(person); // output:
{"name":"John","age":50,"city":"New York"}
```

More about `stringify()`

- Can convert Date to string:

```
var person = {
  name: "John",
  today: new Date()
};

let output = JSON.stringify(person); // {"name":"John","today":"2023-06-
19T08:34:36.988Z"}
```

- `stringify()` will not stringify functions, but it can be fixed if we convert the functions into strings first

```
const person = {
  name: "John",
  age: function () {return 30;}
};
person.age = person.age.toString();

let output = JSON.stringify(person); // output: {"name":"John","age":"function ()
{return 30;}"}
```

- Can stringify (convert to string) an array:

```
const arr = ["John", "Peter", "Sally", "Jane"];

let myString = JSON.stringify(arr); // myString: ["John","Peter","Sally","Jane"]
```

22.5. Object Accessors (getter - setter)

The **reason** for using getters and setters are:

- It gives **simpler** syntax
- It allows equal syntax for properties and methods
- It can **secure better data quality**
- It is useful for doing things **behind-the-scenes**

Object.defineProperty()

This method can also be used to add Getters and Setters

```
// Define object
const obj = {counter : 0};

// Define setters and getters
Object.defineProperty(obj, "reset", {
```

```
    get : function () {this.counter = 0;}
});
Object.defineProperty(obj, "increment", {
    get : function () {this.counter++;}
});
Object.defineProperty(obj, "decrement", {
    get : function () {this.counter--;}
});
Object.defineProperty(obj, "add", {
    set : function (value) {this.counter += value;}
});
Object.defineProperty(obj, "subtract", {
    set : function (value) {this.counter -= value;}
});

// Play with the counter:
// We can treat these getters and setters
// as properties of the object
// So that why we don't need "()" here
obj.reset;
obj.add = 5;
obj.subtract = 1;
obj.increment;
obj.decrement;
```

22.6. Object Constructors

It is considered good practice to **name constructor** functions with an **upper-case first letter**.

```
// this does not have a value, it is a substitute for the new obj
// value of this will become the new obj when a new obj is created
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}
```

Object Types (Blueprints) (Classes)

- We use classes for creating many objects of the same "type".
- We can add a property, a method to an Object.
- We can add a property, a method to a constructor.

Some built-in JavaScript Constructors

```
new String()    // A new String object
new Number()    // A new Number object
new Boolean()   // A new Boolean object
new Object()    // A new Object object
new Array()     // A new Array object
new RegExp()    // A new RegExp object
new Function()  // A new Function object
new Date()      // A new Date object
```

> **Note**: `Math()` object is not in the list. `Math` is a **global** object. The `new` keyword cannot be used on `Math` Primitive values are much faster:
>
> - Use string literals `""` instead of `new String()`.
> - Use number literals `50` instead of `new Number()`.
> - Use boolean literals `true / false` instead of `new Boolean()`.
> - Use object literals `{}` instead of `new Object()`.
> - Use array literals `[]` instead of `new Array()`.
> - Use pattern literals `/()/` instead of `new RegExp()`.
> - Use function expressions `() {}` instead of `new Function()`.

22.7. Object Prototypes

All JavaScript objects **inherit** properties and methods **from a prototype**.

Example:

```
// Object constructor
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyecolor;
}

const myFather = new Person("John", "Doe", 50, "blue");
const myMother = new Person("Sally", "Rally", 48, "green");

// cannot add a new property
// to an existing object constructor
Person.nationality = "English";
```

Using the `prototype` property

Using the keyword `prototype` will allow to add new properties or new methods to object constructors.

```
// Object constructor
function Person(first, last, age, eyecolor) {
```

```
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyecolor;
}

// add new property
Person.prototype.nationality = "English";

// add new method
Person.prototype.name = function() {
  return this.firstName + " " + this.lastName;
};
```

22.8. JavaScript Iterables

Iterating over a String

```
for (const x of "W3Schools") {
  console.log(x);
}
```

Iterating over an Array

```
for (const x of [1,2,3,4,5]) {
  console.log(x);
}
```

JavaScript Iterators

- Home-made Iterable: Cannot use with `for...of` statement

Example:

```
// Home Made Iterable
function myNumbers() {
  let n = 0;
  return {
    next: function() {
      n += 10;
      return {value:n, done:false};
    }
  };
}

// Create Iterable
```

```
// Never ending every time next() is called
const n = myNumbers();
n.next(); // Returns 10
n.next(); // Returns 20
n.next(); // Returns 30
```

- Other type of Iterator: A **JavaScript iterable** is an object that has a `Symbol.iterator`.

Example:

```
// Create an Object
myNumbers = {};

// Make it Iterable
myNumbers[Symbol.iterator] = function() {
  let n = 0;
  done = false;
  return {
    next() {
      n += 10;
      if (n == 100) {done = true}
      return {value:n, done:done};
    }
  };
}

// Symbol.iterator is called automatically by for...of
for (const num of myNumbers) {
  // Any Code Here
  console.log(num);
}
```

22.9. JavaScript Sets

A collection of **unique** values (it can hold any value of **any data type**), each can only occur **once** in a set.

Method:

Method	Description
`new Set()`	Create a new Set
`add()`	Add a new element to the Set
`delete()`	Remove an element from the Set
`has()`	return true if a value exists
`clear()`	remove all elements from the Set
`forEach()`	invoke a callback for each element

Method	Description
`values()`	return an Iterator with all values in the Set
`keys()`	same as `values()`, make it compatible with Maps
`entries()`	return an Iterator with `[value,value]` pairs from a Set

Property:

Property	Description
size	Return number of elements

Example:

```
// Create Variables
const a = "a";
const b = "b";
const c = "c";

// Create a Set
const letters = new Set();

// Add Variables to the Set
letters.add(a);
letters.add(b);
letters.add(c);

// List all entries
let text = "";
letters.forEach (function(value) {
  text += value;
})

// Create an Iterator using values()
const myIterator = letters.values();

// List all Values
let text1 = "";
for (const entry of myIterator) {
  text1 += entry;
}
```

22.10. JavaScript Maps

- Holds **key-value pairs** where the keys can be **any datatype**
- Remembers the **original insertion order** of the keys
- Has a property that represnets the **size** of the map.

Method:

Method	Description
`new Map()`	Create a new Map
`set("key","value")`	Set the value for a key in a Map
`get("key")`	Get the value for a key in a Map
`delete("key")`	Remove a Map element specified by a key
`has("key")`	Return true if a key exists
`clear()`	Remove all elements from the Map
`forEach()`	Invoke a callback for each key/value pair
`values()`	Return an Iterator with all values in the Map
`keys()`	Return an Iterator with the keys in a Map
`entries()`	Return an Iterator with `[key,value]` pairs from a Map

Property:

Property	Description
`size`	Return number of elements

Example:

```
// Create a Map
const fruits = new Map();

// Set Map Values
fruits.set("apples", 500);
fruits.set("bananas", 300);
fruits.set("oranges", 200);

// List all entries
let text = "";
fruits.forEach (function(value, key) {
  text += key + ' = ' + value;
})

// List all entries
let text1 = "";
for (const x of fruits.entries()) {
  text1 += x;
}
```

Object as Keys

```
// Create Objects
const apples = {name: 'Apples'};
const bananas = {name: 'Bananas'};
const oranges = {name: 'Oranges'};

// Create a Map
const fruits = new Map();

// Add new Elements to the Map
fruits.set(apples, 500);
fruits.set(bananas, 300);
fruits.set(oranges, 200);

// Access to a value of key
fruits.get(apples);
```

Compare between Objects and Maps

Object	Map
Not directly iterable	Directly iterable
Do not have a size property	Have a size property
Keys must be **Strings** (or Symbol)	Keys can be **any datatype**
Keys are not well ordered	Keys are ordered by insertion
Have default keys	Do not have default keys

22.11. Object Methods

Managing Objects

```
// Create object with an existing object as prototype
Object.create()

// Adding or changing an object property
Object.defineProperty(object_name, property, descriptor)

// Adding or changing object properties
Object.defineProperties(object_name, descriptors)

// Accessing Properties
Object.getOwnPropertyDescriptor(object_name, property)

// Returns all properties as an array
Object.getOwnPropertyNames(object_name)

// Accessing the prototype
Object.getPrototypeOf(object_name)
```

```
// Returns enumerable properties as an array
Object.keys(object_name)
```

Protecting Objects

```
// Prevents adding properties to an object
Object.preventExtensions(object_name)

// Returns true if properties can be added to an object
Object.isExtensible(object_name)

// Prevents changes of object properties (not values)
Object.seal(object_name)

// Returns true if object is sealed
Object.isSealed(object_name)

// Prevents any changes to an object
Object.freeze(object_name)

// Returns true if object is frozen
Object.isFrozen(object_name)
```

Changing Meta Data

```
// Property value can/cannot be changed (read-only)
writable : true
// Property can be enumerated
enumerable : true
// Property can be reconfigured
configurable : true
```

23. Classes

23.1. Introduction

- Use the keyword `class` to create a class.
- Always add a method named `constructor()`
- **Class methods** are created with the same syntax as **object methods**

Example

```
class Car {
  constructor(name, year) {
    this.name = name;
```

```
    this.year = year;
  }
  age() {
    const date = new Date();
    return date.getFullYear() - this.year;
  }
}

const myCar = new Car("Ford", 2014);
document.getElementById("demo").innerHTML =
"My car is " + myCar.age() + " years old.";
```

"use strict"

In **"strict mode"** you will get an error if you use a variable without declaring it

23.2. Class Inheritance

- To create a **class inheritance**, use the `extends` keyword, it will inherits all the methods from another class
- Use getters and setters for your properties
- Class declaration are **not hoisted** (must de declared before being used).

Example

```
class Car {
    constructor(brand) {
        this._carname = brand;
    }
    present() {
        return 'I have a ' + this.carname;
    }
    // use get and set keyword to create setters and getters
    get carname() {
        return this._carname;
    }
    set carname(x) {
        this._carname = x;
    }
}

// super() refers to parent class
// when called in constructor, we call the parent's constructor
// and get access to the parent's properties and methods
class Model extends Car {
    constructor(brand, mod) {
        super(brand);
        this.model = mod;
    }
    show() {
        return this.present() + ', it is a ' + this.model;
```

```
    }
}

const myCar = new Car("Ford");
myCar.carname = "Volvo";
document.getElementById("demo").innerHTML = myCar.carname;
```

> **Note**:
>
> - Even if the **getter** is a method, you do **not use parentheses** when you want to get the property value.
> - Use the **underscore** character to **separate** the getter/setter **from** the **actual property**
> - To use a **setter**, use the same syntax as when you set a property value, **without parentheses**

23.3. JavaScript Static Methods

> **Note**:
>
> - Static class methods are defined on the class itself.
> - You **cannot call** a static method on an object, only **on an object class**.

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello() {
    return "Hello!!";
  }
}

const myCar = new Car("Ford");

// You can call 'hello()' on the Car Class:
document.getElementById("demo").innerHTML = Car.hello();

// But NOT on a Car Object:
// document.getElementById("demo").innerHTML = myCar.hello();
// this will raise an error.
```

24. JavaScript Async

24.1. JavaScript Callbacks

Example

```
// Create an Array
const myNumbers = [4, 1, -20, -7, 5, 9, -6];
```

```
// Call removeNeg with a callback
// When passing func as argument, do not using ()
const posNumbers = removeNeg(myNumbers, (x) => x >= 0);

// Display Result
document.getElementById("demo").innerHTML = posNumbers;

// Keep only positive numbers
function removeNeg(numbers, callback) {
  const myArray = [];
  for (const x of numbers) {
    if (callback(x)) {
      myArray.push(x);
    }
  }
  return myArray;
}
```

In this example, `(x) => x >= 0` is a callback function and it is passed to `removeNeg()` as an **argument**.

24.2. Asynchronous

- Functions running in parallel with other functions are called asynchronous.
- Callback in the last chapter is often used with asynchronous functions.

Waiting for a Timeout

Specify a callback function to be executed on time-out

```
// 3000 is the number of milliseconds before time-out
setTimeout(function() { myFunction("I love You !!!"); }, 3000);

function myFunction(value) {
  document.getElementById("demo").innerHTML = value;
}
```

Waiting for Intervals

Specify a callback function to be executed for each **interval**

```
setInterval(myFunction, 1000);

function myFunction() {
  let d = new Date();
  document.getElementById("demo").innerHTML=
  d.getHours() + ":" +
  d.getMinutes() + ":" +
```

```
  d.getSeconds();
}
```

24.3. Promises

- **"Producing code"** is code that can **take** some time
- **"Consuming code"** is code that must **wait** for the result
- A **Promise** is a JavaScript object that **links** producing code and consuming code
- A JavaScipt Promise object has 2 properties: **state** and **result**, we cannot access the properties.
- A JavaScipt Promise object can be:
    - Pending (working), result is undefined
    - Fulfilled, reulst is a value
    - Rejected, result is an error object

```
let myPromise = new Promise(function(myResolve, myReject) {
// "Producing Code" (May take some time)

  myResolve(); // when successful
  myReject();  // when error
});

// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

25. HTML DOM

25.1. Introduction

DOM: Document Object Model

- DOM defined a standard for accessing documents
- HTML DOM is a standard object model and programming **interface** for HTML.
    - The HTML elements as **objects**
    - The **properties** of all HTML elements
    - The **methods** to access all HTML elements
    - The **events** for all HTML elements

Use this for to get, change, add or delete HTML elements.

25.2. HTML DOM methods

- **HTML DOM methods** are **actions** you can perform (on HTML Elements).
- **HTML DOM properties** are **values** (of HTML Elements) that you can set or change.
- In the DOM, all HTML elements are defined as objects.

`getElementById` method used `id="id_name"` to find the element

`innerHTML` property can be used to get or change any HTML element, including `<html>` and `<body>`

25.3. HTML DOM Document

It is the owner of all other objects in your web page

Finding HTML elements

Method	Description
`document.getElementById(id)`	Find an element by element **id**
`document.getElementByTagName(name)`	Find an element by **tag** name
`document.getElementByClassName(name)`	Find an element by **class** name

Changing HTML Elements

Property	Description
`element.innerHTML = new html content`	Change the inner HTML of an element
`element.attribute = new value`	Change the attribute value of an HTML element
`element.style.property = new style`	Change the style of an HTML element

Method	Description
`element.setAttribute(attribute, value)`	Change the attribute value of an HTML element

Adding and Deleting Elements

Method	Description
`document.createElement(element_name)`	Create an element
`document.removeChild(element_name)`	Remove an element
`document.appendChild(element_name)`	Add an element
`document.replaceChild(new, old)`	Replace an element
`document.write(text)`	Write into the HTML output stream

Adding Events Handlers

Method	Description
`document.getElementById(id).onclick = function() {code}`	Adding event handler code to an `onclick` event

Finding HTML Objects

Property	Description	DOM
`document.anchors`	Returns all `<a>` elements that have a name attribute	1
`document.applets`	**Deprecated**	1
`document.baseURI`	Returns the absolute base URI of the document	3
`document.body`	Returns the `<body>` element	1
`document.cookie`	Returns the document's cookie	1
`document.doctype`	Returns the document's doctype	3
`document.documentElement`	Returns the `<html>` element	3
`document.documentMode`	Returns the mode used by the browser	3
`document.documentURI`	Returns the URI of the document	3
`document.domain`	Returns the domain name of the document server	1
`document.domConfig`	Obsolete.	3
`document.embeds`	Returns all `<embed>` elements	3
`document.forms`	Returns all `<form>` elements	1
`document.head`	Returns the `<head>` element	3
`document.images`	Returns all `<img>` elements	1
`document.implementation`	Returns the DOM implementation	3
`document.inputEncoding`	Returns the document's encoding (character set)	3
`document.lastModified`	Returns the date and time the document was updated	3
`document.links`	Returns all `<area>` and `<a>` elements that have a `href` attribute	1
`document.readyState`	Returns the (loading) status of the document	3
`document.referrer`	Returns the URI of the referrer (the linking document)	1
`document.scripts`	Returns all `<script>` elements	3
`document.strictErrorChecking`	Returns if error checking is enforced	3
`document.title`	Returns the `<title>` element	1
`document.URL`	Returns the complete URL of the document	1

25.4. HTML DOM Elements

Finding HTML Elements

- Finding HTML elements by **id** `document.getElementById(id)`
- Finding HTML elements by **tag** name `document.getElementByTagName(tag_name)`
- Finding HTML elements by **class** name `document.getElementByClassName(class_name)`
- Finding HTML elements by **CSS selectors** `document.querySelectorAll(element)`
- Finding HTML elements by **HTML object collections** `document.anchors document.body document.documentElement document.embeds document.forms document.head document.images document.links document.scripts document.title`

25.5. HTML DOM - Changing HTML

Changing HTML Content

```
document.getElementById(id).innerHTML = newHTMLContent
```

Changing the value of an Attribute

```
document.getElementById(id).attribute = new value
```

25.6. JavaScript Forms

```
<input id="numb">

<button type="button" onclick="myFunction()">Submit</button>

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

<script>
function myFunction() {
  // Get the value of the input field with id="numb"
  let x = document.getElementById("numb").value;
  // If x is Not a Number or less than one or greater than 10
  let text;
  if (isNaN(x) || x < 1 || x > 10) {
    text = "Input not valid";
  } else {
    text = "Input OK";
  }
  document.getElementById("demo").innerHTML = text;
}
</script>
```

Data Validation

Data validation is the process of ensuring that **user input** is **clean**, **correct**, and **useful**.

Typical validation tasks are:

- has the user filled in **all required fields**?
- has the user entered a **valid date**?
- has the user entered text in a **numeric field**?

Most often, the purpose of data validation is to **ensure correct user input**.

Validation can be defined by many different methods, and deployed in many different ways.

- **Server side validation** is performed by a web server, after input has been sent to the server.
- **Client side validation** is performed by a web browser, before input is sent to a web server.

25.7. HTML DOM - Changing CSS

Changing HTML Style

```
document.getElementById(id).style.property = new style
```

Using Events

Events are generated by the browser when "**things happen**" to HTML elements:

- An element is clicked on
- The page has loaded
- Input fields are changed

Example

```
<h1 id="id1">My Heading 1</h1>

<!--Change the style when the user click-->
<button type="button"
onclick="document.getElementById('id1').style.color = 'red'">
Click Me!</button>
```

```
<!DOCTYPE html>
<html>
<!--
The container with style = "position: relative".
The animation with style = "position: absolute".
-->
<style>
#container {
  width: 400px;
  height: 400px;
  position: relative;
  background: yellow;
}
```

```
#animate {
  width: 50px;
  height: 50px;
  position: absolute;
  background-color: red;
}
</style>
<body>

<p><button onclick="myMove()">Click Me</button></p>

<div id ="container">
  <div id ="animate"></div>
</div>

<script>
function myMove() {
  let id = null;
  const elem = document.getElementById("animate");
  let pos = 0;
  clearInterval(id);
  id = setInterval(frame, 5);
  function frame() {
    if (pos == 350) {
      clearInterval(id);
    } else {
      pos++;
      elem.style.top = pos + "px";
      elem.style.left = pos + "px";
    }
  }
}
</script>

</body>
</html>
```

25.8. HTML DOM Events

Examples of **HTML events**:

- When a user **clicks** the mouse

```
<h1 onclick="changeText(this)">Click on this text!</h1>

<script>
function changeText(id) {
  id.innerHTML = "Ooops!";
}
</script>
```

- When a web page has **loaded**
- When an image has been **loaded**
    - The `onload` and `onunload` events are triggered when the user **enters** or **leaves** the **page**.
    - They can be used to deal with **cookies**
    - `<body onload="checkCookies()">`
- When the mouse **moves over** an element

```
<div onmouseover="mOver(this)" onmouseout="mOut(this)">Mouse Over Me</div>

<script>
function mOver(obj) {
  obj.innerHTML = "Thank You"
}

function mOut(obj) {
  obj.innerHTML = "Mouse Over Me"
}
</script>
```

    - `onmousedown`, `onmouseup`, and `onclick` events are all parts of a **mouse-click**.

```
<div onmousedown="mDown(this)" onmouseup="mUp(this)">Click Me</div>

<script>
function mDown(obj) {
  obj.style.backgroundColor = "#1ec5e5";
  obj.innerHTML = "Release Me";
}

function mUp(obj) {
  obj.style.backgroundColor="#D94A38";
  obj.innerHTML="Thank You";
}
</script>
```

- When an input field is **changed**
- When an HTML form is **submitted**
- When a user **strokes a key**

Add many event handlers to the same element

`addEventListener()` allow you to add many events to the same element, without overwtiting the existing events

```
element.addEventListener("mouseover", myFunction);
element.addEventListener("click", mySecondFunction);
element.addEventListener("mouseout", myThirdFunction);
```

Event Capturing or Event Bubbling

There are two ways of **event propagation** in the HTML DOM, **bubbling** and **capturing**.

> **Event propagation** is a way of defining the element order when an event occurs. If you have a `<p>` element inside a `<div>` element, and the user clicks on the `<p>` element, which element's "click" event should be handled first?

- In **bubbling** the **inner most** element's event is handled **first** and then the outer: the `<p>` element's click event is handled first, then the `<div>` element's click event.
- In **capturing** the **outer most** element's event is handled **first** and then the inner: the `<div>` element's click event will be handled first, then the `<p>` element's click event.

With the `addEventListener()` method you can specify the propagation type by using the "**useCapture**" parameter:

```
addEventListener(event, function, useCapture);
```

Remove event

`removeEventListener()` removes event handlers that have been attached with the `addEventListener()`

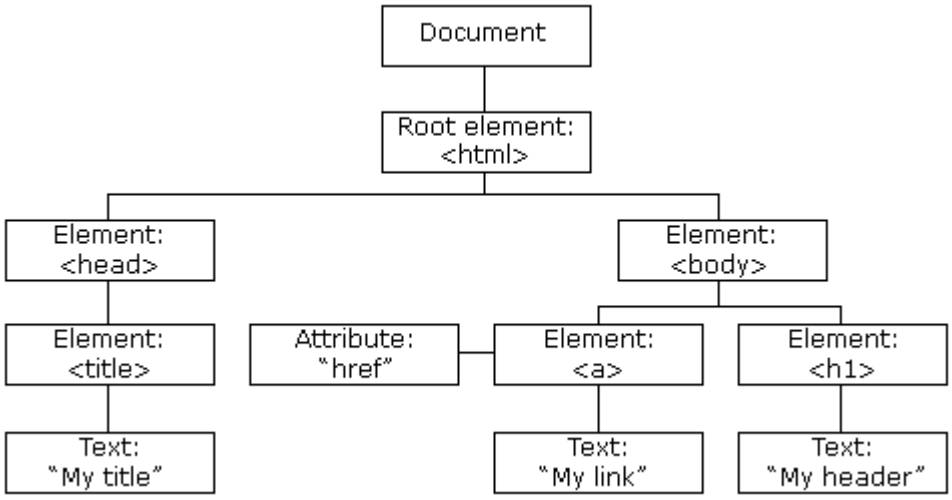
```
element.removeEventListene ("mousemove", myFunction);
```

25.9. HTML DOM Navigation

DOM Nodes

According to the **W3C HTML DOM** standard, everything in an HTML document is a node:

- The entire document is a **document node**
- Every HTML element is an **element node**
- The text inside HTML elements are **text nodes**
- Every HTML attribute is an **attribute node** (deprecated)
- All comments are **comment nodes**



Navigation between Nodes

You can use the following node **properties** to navigate between nodes with JavaScript:

- `parentNode`
- `childNodes[nodenumber]`
- `firstChild`
- `lastChild`
- `nextSibling`
- `previousSibling`
- `document.body` - body of the document
- `document.documentElement` - full document

Example

```
// Accessing the first child
myTitle = document.getElementById("demo").childNodes[0].nodeValue;
```

```
// Retrieve the text of <h1>, copy it to <p>
<h1 id="id01">My First Page</h1>
<p id="id02"></p>

<script>
document.getElementById("id02").innerHTML =
document.getElementById("id01").innerHTML;
</script>
```

The nodeName Property

The **nodeName** property specifies the name of a node.

- **nodeName** is **read-only**

- **nodeName** of an element node is the same as the tag name
- **nodeName** of an attribute node is the attribute name
- **nodeName** of a text node is always #text
- **nodeName** of the document node is always #document

> 📝**Note**: **nodeName** always contains the uppercase tag name of an HTML element.

The nodeValue Porperty

The **nodeValue** property specifies the value of a node.

- **nodeValue** for element nodes is `null`
- **nodeValue** for text nodes is the text itself
- **nodeValue** for attribute nodes is the attribute value

The nodeType Property

The `nodeType` property is **read only**. It returns the type of a node.

25.10. HTML DOM Elements (Nodes)

Creating new HTML Elements

Append new element (node) as the last child of `element_parent`

```
element_parent.appendChild(new_element);
```

Example:

```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script>
// Creating new <p> element
const para = document.createElement("p");
// add text to <p> element
const node = document.createTextNode("This is new.");
// add new element to an existing element
para.appendChild(node);

const element = document.getElementById("div1");
element.appendChild(para);
</script>
```

Creating new HTML Elements - `insertBefore()`

Append `element_child` as a children of `element_parent`, be the children before the `element_target`.

```
element_parent.insertBefore(element_target, element_child);
```

Example:

```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script>
const para = document.createElement("p");
const node = document.createTextNode("This is new.");
const element = document.getElementById("div1");
const child = document.getElementById("p1");
element.insertBefore(para, child);
</script>
```

Removing Existing HTML elements

Use `remove()` or `removeChild()`

```
const elmnt = document.getElementById("p1"); elmnt.remove();
```

```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script>
const parent = document.getElementById("div1");
const child = document.getElementById("p1");
parent.removeChild(child);
</script>
```

Replacing HTML elements

Use `replaceChild()`

```
parent_element.replaceChild(new_element, child_element);
```

25.11. HTML DOM Collections

HTML Collection Object

- The `getElementsByTagName()` method returns an `HTMLCollection` object.
- An `HTMLCollection` object is an **array-like list (collection)** of HTML elements.
- `length` property defines the number of elements

Example:

```
// sellect all <p> elements
const myCollection = document.getElementsByTagName("p");

// access the second <p> element
myCollection[1];
```

> **Note**: An HTML Collection is NOT an Array, look like but NOT.

25.12. HTML DOM Node Lists

- A `NodeList` object is a list (collection) of nodes extracted from a document.
- A `NodeList` object is almost the same as an `HTMLCollection` object.
- Use `querySelectorAll()` to return a NodeList object

Difference between an HTMLCollection and a NodeList

- Both are much the same thing, **array-like collections (lists)** of nodes (elements) extracted from a document, nodes can be accessed using index (start at 0).
- Both have `length` property
- `HTMLCollection` items can be accessed by their **name**, **id**, or **index** number and it is a **live** collection (when adding a new element, list in the `HTMLCollection` also change)
- `NodeList` items can only be accessed by their **index** number and it is a **static** collection (when adding a new element, list in `NodeList` not change)
- The `getElementsByClassName()` and `getElementsByTagName()` methods return a live `HTMLCollection`.
- The `querySelectorAll()` method returns a **static** `NodeList`.
- The `childNodes` property returns a **live** `NodeList`.