JS (JavaScript)

JS

- JS Introduction
- JS Where To
- JS Output
- JS Syntax
- JS Variables
- JS Let
- JS Const
- JS Comments
- JS Statements
- JS Data Types
- JS Functions
- JS Objects
- JS Events
- JS Strings
- JS String Methods
- JS String Search
- JS String Templates
- JS Numbers
- JS Number Methods
- JS Conditions
- JS Switch
- JS Loop For
- JS Loop For In
- JS Loop For of
- JS Loop While
- JS Arrays
- JS Array Methods
- JS Array Sort
- JS Array Iteration
- JS Array Const
- JS Dates
- JS Date Formats
- JS Math
- JS Booleans
- JS Comparisons
- JS Type of and Type Conversion
- JS this Keyword & Arrow Function

JS Introduction

- JavaScript is the world’s most popular programming language.
- JavaScript is The programming language of the Web.
- JavaScript is easy to learn.

One of many JavaScript HTML methods is getElementById().

E.g.

```
document.getElementById(“hello”).innerHTML = “Hello JavaScript”;
```

Note: JavaScript can change HTML Content, HTML Attribute Values, HTML Styles, Show and Hide of HTML Elements .

JS Where To

In HTML, JavaScript code is inserted between <script> and </script> tags.

A JavaScript function is a block of JavaScript code, that can be executed when “called” for.

For example, a function can be called when an event occurs, like when the use clicks a button.

Scripts can be placed in the <body>, or in the <head> section of an HTML page, or in both.

Scripts can also be placed in external files.

External JavaScript Advantages

Placing scripts in external files has some advantages :

- It separates HTML and code
- It makes HTML and JavaScript easier to read and maintain
- Cached JavaScript files can speed up page loads

External References

An external script can be referenced in 3 different ways:

- With a full URL (a full web address)
- With a file path (like /js/)
- Without any path

JS Output

JavaScript can “display” data in different ways:

- Writing into an HTML element, using innerHTML.
- Writing into the HTML output using document.write().
- Writing into an alert box, using window.alert().
- Writing into the browser console, using console.log().

E.g.

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

<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>
```

E.g.

```
<script>
document.write(5 + 6);
</script>
```

Note: document.write() method should only be used for testing.

E.g.

```
<script>
window.alert(5 + 6);
</script>
```

```
<script>
alert(5 + 6);
</script>
```

E.g.

```
<script>
console.log(5 + 6);
</script>
```

F12 on your keybord will activate debugging.

Then select "Console" in the debugger menu.

JS Syntax

The JavaScript syntax defines two types of values :

- Fixed values
- Variable values

Fixed values are called Literals.

Variable values are called Variables.

JavaScript and Camel Case

Historically, programmers have used different ways of joining multiple words into one variable name:

- Hyphens : first-name, last-name (Hyphens are not allowed in JS.)
- Underscore: first_name, last_name
- Upper Camel Case (Pascal Case): FirstName, LastName
- Lower Camel Case : firstName, lastName

JavaScript Literals

The two most important syntax rules for fixed values are

1. Numbers are written with or without decimals.
2. Strings are text, written within double or single quotes.

E.g.

Numbers : 10.50 , 1001

Strings : “John Doe”, ‘John Doe’

JS Variables

There are 3 ways to declare a JavaScript variable:

- Using var
- Using let
- Using const

Variables

Variables are containers for storing data (values).

All JavaScript variables must be identified with unique names.

These unique names are called identifiers.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, total).

The general rules for constructing names for variables (unique identifiers) are :

- Name can contain letters, digits, underscores, and dollar signs.
- Name must begin with letter
- Names can also begin with $ and _ (but we will not use it in this tutorial)
- Names are case sensitive (y and Y are different variables)
- Reserved words (like JavaScript keywords) cannot be used as names

E.g.

```
var x = 5
var $money = 500
var sum = x + $money
```

JS Let

The let keyword was introduced in ES6 (2015).

- Variables defined with let cannot be Redeclared.
- Variables defined with let must be Declared before use.
- Variable defined with let have Block Scope.

E.g.

```
let x = "John Doe";

let x = 0;

// SyntaxError: 'x' has already been declared
```

E.g.

```
//Block Scope

        let x = 10;

{
        let x = 2;

}
```

E.g.

```
carName = "Saab";
let carName = "Volvo";

ReferenceError : let must declare before use
```

JS Const

The const keyword was introduced in ES6 (2015).

- Variables defined with const cannot be Redeclared.
- Variables defined with const cannot be Reassigned.
- Variables defined with const have Block Scope.

E.g.

```
const PI = 3.141592653589793;
PI = 3.14;      // This will give an error
PI = PI + 10;   // This will also give an error
```

When to use JavaScript const ?

As a general rule, always declare a variable with const unless you know that the value will change.

Use const when user declare:

- A new Array
- A new Object
- A new Function
- A new RegExp

E.g.

```
const pi = 3.142

pi = 3.14 // Error
```

E.g.

```
var x = 2;     // Allowed
const x = 2;   // Not allowed

{
let x = 2;     // Allowed
const x = 2;   // Not allowed
}

{
const x = 2;   // Allowed
const x = 2;   // Not allowed
}
```

JS Comments

JavaScript comments can be used to explain JavaScript code, and to make it more readable.

JavaScript comments can also be used to prevent execution , when testing alternative code.

Single Line Comments (//)

Multi-Line Comments (/* */)

JS Statements

A computer program is a list of “ instructions “ to be “ executed” by a computer.

In a programming language, these programming instructions are called statements.

Semicolon separate JavaScript statements.
Add a semicolon at the end of each executable statement.

JavaScript ignores multiple spaces. User can add white space to your script to make it more readable.

E.g.

```
let x, y, z;          // statement 1 ( Declare 3 variables )

x = 5;                // statement 2 ( Assign the value 5 to x )
y = 6;                // statement 3  ( Assign the value 6 to y )
z = x + y;            // statement 4 ( Assign the sum of x and y to z )
```

JS Operators

JavaScript Arithmetic Operators

Arithmetic operators are used to perform arithmetic on numbers:

Operator	**Description**
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (ES2016)
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

JavaScript Assignment Operators

Assignment operators assign values to JavaScript variables.

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

JavaScript comparison Operators

Operator	**Description**
==	Equal to
===	Equal value and equal type
!=	Not equal
!==	Not equal value or not equal type
>	Greater than
<	Less than
>=	Greater than equal to
<=	Less than or equal to
?	Ternary operator

JavaScript Logical Operators

Operator	Description
&&	Logical and
||	Logical or
!	Logical not

JavaScript Type Operators

Operator	Description
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

JS Data Types

JavaScript variables can hold different data types: numbers, stings, objects and more.

In Programming, data types is an important concept.
To be able to operate on variables, it is important to know something about the type.

Without data types, a computer cannot safely solve.

E.g.

```
let x                                              // Undefined
let y = null;                                      // Null
let z = true;                                      // Booleans
let length = 16;                                   // Number
let lastName = “Johnson”;                          // String
let x = { firstName: “John”, lastName : “Doe” };   // Object
```

JS Functions

A JavaScript function is defined with the function keyword, followed by a name, followed by parentheses ( ).

Function names can contain letters, digits, underscores and dollar signs (same rules as variables).

There have four types of Functions (method)

- no return function (method)
- return function (method)
- parameter function (method)
- return parameter function (method)

no return function (method)

E.g.

```
function myfunction ( ) {
        alert ( " Hello World " ) ;
}
```

return function (method)

E.g.

```
function myfunction ( ) {
        return 1 + 2 ;
}
```

parameter function (method)

E.g.

```
function myfunction (a , b ) {
        alert ( a + b ) ;
}
```

return parameter function (method)

E.g.

```
function myfunction (a , b ) {
        return a + b ;
}
```

JS Objects

In real life, a car is an object.

A car has properties like weight and color , and methods like drive.
All cars have the same properties, but the property values differ from car to car.
All cars have the same methods, but the methods are performed at different times.

Object	Properties	Methods
	car.name = Fiat	car.start()
	car.model = 500	car.drive()
	car.weight = 850kg	car.brake()
	car.color = white	car.stop()

The values are written as **name:value** pairs (name and value separated by a colon).

It is a common practice to declare objects with the const keyword.

E.g.

const person = { firstName : “John”, lastName : “Doe”, age : 50 }

Accessing Object Properties

User can access object properties in two ways :

1. objectName.propertyName
2. objectName[”propertyName”]

Object Methods

Object can also have methods.

Methods are actions that can be performed on objects.
Methods are stored in properties as function definitions.

E.g.

```
const person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

The this Keyword

In a function definition, this refers to the “owner” of the function.

In other words, this.firstName means the firstName property of the **this object**.

Accessing Object Methods

Syntax:

objectName.methodName()

E.g.

name = person.fullName();

JS Events

HTML events are “**things**” that happen to HTML elements.

When JavaScript is used in HTML pages, JavaScript can “**react**” on these events.

An HTML event can be something the browser does, or something a user does.

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

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

JS Strings

JavaScript strings are used for storing and manipulating text.

A JavaScript string is zero or more characters written inside quotes.

To find the length of a string, use the built-in length property.

E.g.

```
let x = “I’m ok”;
let y = ”I am ok”;
let z = “He is ‘Johnny’”
```

Escape Character

Because string must be written within quotes, JavaScript will misunderstand this string.

E.g.

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

Code	**Result**	**Description**
\’	‘	Single quote
\”	“	Double quote
\\	\	Backslash

Strings Can be Objects

Normally, JavaScript strings are primitive values, created from literals:

let firstName = “John”;

But strings can also be defined as objects with the keyword new:

let firstName = new String(“john”);

Note: Don’t create strings as objects. It slows down execution speed. The new keyword can complicate the code and produce unexpected results.

JS String Methods

String methods help user to work with strings.

String Methods and Properties

Primitive values, like “John Doe”, cannot have properties or methods (because they are not objects).

But with JavaScript, methods and properties are also available to primitive values, because JavaScript treats primitive values as objects when executing methods and properties.

Extracting String Parts

There are 3 methods for extracting a part of a string:

- slice(start, end)
- substring(start, end)
- substr(start, length)

The slice() Method

slice() extracts a part of a string and returns the extracted part in a new string.

The method takes 2 parameters : the start position, and the end position (end not included).

This example slices out a portion of a string from position 7 to position 12.

E.g.

```
let str = "Apple, Banana, Kiwi";
let part = str.slice(7, 13);
```

Note : JavaScript counts positions from zero. First position is 0.

If a parameter is negative, the position is counted from the end of the string.

If you omit the second parameter, the method will slice out the rest of the string, or counting from the end.

The substring() Method

substring() is similar to slice().

The difference is that substring() cannot accept negative indexes.

E.g.

```
let str = "Apple, Banana, Kiwi";
let part = substring(7, 13);
```

If user omit the second parameter, substring() will slice out the rest of the string.

The substr() Method

substr() is similar to slice().

The difference is that the second parameter specifies the length of the extracted part.

E.g.

```
let str = "Apple, Banana, Kiwi";
let part = str.substr(7, 6);

let str = "Apple, Banana, Kiwi";
let part = str.substr(7);
```

Replacing String Content

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

E.g.

```
let text = "Please visit Microsoft!";
let newText = text.replace("Microsoft", "W3Schools");
```

Note: The replace() method does not change the string it is called on. It returns a new string.
By default, the replace() method replaces only the first match. The replace() method is case sensitive.

To replace case insensitive, use a regular expression with an ” /i ” flag (insensitive).

To replace all matches, use a regular expression with a /g flag (global match).

Converting to Upper and Lower Case

A string is converted to upper case with toUpperCase().

E.g.

```
let text1 = "Hello World!";
let text2 = text1.toUpperCase();
```

A string is converted to lower case with toLowerCase().

E.g.

```
let text1 = "Hello World!";
let text2 = text1.toLowerCase();
```

The concat() Method

concat() joins two or more strings.

E.g.

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

The concat() method can be used instead of the plus operator. These two lines do the same.

Note: All string methods return a new string. They don’t modify the original string.

Formally said: Strings are immutable: Strings cannot be changed, only replaced.

String.trim()

The trim() method removes whitespace from both sides of a string.

E.g.

```
let text = "       Hello World!        ";
text.trim()    // Returns "Hello World!"
```

JavaScript String Padding

ECMAScript 2017 added two String methods: padStart and padEnd to support padding at the beginning and at the end of a string.

E.g.

```
let text = "5";
text.padStart(4,0)    // Returns 0005
```

```
let text = "5";
text.padEnd(4,0)     // Returns 5000
```

Extracting String Characters

There are 3 methods for extracting string characters:

- charAt (position)
- charCodeAt (position)
- Property access [ ]

The charAt() Method

The charAt() method returns the character at a specified index (position) in a string.

E.g.

```
let text = "HELLO WORLD";
text.charAt(0)          // Returns H
```

The charCodeAt() Method

The charCodeAt() method returns the Unicode of the character at a specified index in a string.

E.g.

```
let text = "HELLO WORLD";

text.charCodeAt(0)      // Returns 72
```

Property Access

ECMAScript 5 (2009) allows property access [ ] on strings.

E.g.

```
let text = "HELLO WORLD";
text[0]                 // returns H
```

Converting a String to an Array

A string can be converted to an array with the split() method.

E.g.

```
text.split(“,”);
text.split(“”);
```

JS String Search

JavaScript methods for searching strings.

- String.indexOf()
- String.lastIndexOf()
- String.startsWith()
- String.endsWith()

String.indexOf()

The indexOf() method returns the index of (the position of) the first occurrence of a specified text in a string.

E.g.

```
let str = "Please locate where 'locate' occurs!";
str.indexOf("locate");
```

String.lastIndexOf()

The lastIndexOf() method returns the index of the last occurrence of a specified text in a string.

E.g.

```
let str = "Please locate where 'locate' occurs!";
str.lastIndexOf("locate");
```

Note: Both indexOf() and lastIndexOf() return -1 if the text is not found.

String.search()

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

E.g.

```
let str = "Please locate where 'locate' occurs!";
str.search("locate");
```

The two methods, indexOf() and search() , are equal ?

They accept the same arguments (parameters), and return the same value ?

The two methods are NOT equal. These are the differences:

- The search() method cannot take a second start position argument.
- The indexOf() method cannot take powerful search values (regular expressions).

String.match()

The match() method search a string for a match against a regular expression, and returns the matches, as an Array object.

E.g.

```
let text = "The rain in SPAIN stays mainly in the plain";
text.match(/ain/g);
```

String.includes()

The includes() method returns true if a string contains a specified value.

E.g.

```
let text = "Hello world, welcome to the universe.";
text.includes("world");  // Return ” true “
```

String.startsWith()

The startsWith() method returns true if a string begins with a specified value, otherwise false.

E.g.

```
let text = "Hello world, welcome to the universe.";

text.startsWith("world", 6)    // Returns true
```

Note: The startsWith() method is case sensitive.

String.endsWith()

The endsWith() method returns true if a string ends with a specified value, otherwise false.

E.g.

```
var text = "John Doe";
text.endsWith("Doe");
```

JS String Templates Literals

Synonyms :

- Template Literals
- Template Strings
- String Templates
- Back-Tics Syntax

Back-Tics Syntax

Template Literals use back-ticks (``) rather than the quotes (“”) to define a string.

let text = `Hello World!` ;

Quotes Inside Strings

With template literals, user can use both single and double quotes inside a string.

E.g.

```
let text = `He's often called "Johnny” `;
```

Multiline Strings

Template literals allows multiline strings.

E.g.

```
let text =
`The quick
brown fox
jumps over
the lazy dog`;
```

Interpolation

Template literals provide an easy way to interpolate variables and expressions into strings.

The method is called string interpolation.

The syntax is :

${.....}

Variable Substitutions

Template literals allow variables in strings.

E.g.

```
let firstName = "John";
let lastName = "Doe";

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

JS Numbers

JavaScript has only one type of number. Numbers can be written with or without decimals.

Extra large or extra small numbers can be written with scientific (exponent) notation.

E.g.

```
let x = 3.14;    // A number with decimals
let y = 3;       // A number without decimals

let a = 123e5;    // 12300000
let b = 123e-5;   // 0.00123
```

JavaScript Numbers are Always 64-bit Floating Point

Unlike many other programming languages, JavaScript does not define different types of numbers, like integers, short, long, floating-point etc.

JavaScript numbers are always stored as double precision floating point number, flowing the international IEEE 754 standard.

This format stores numbers in 64 bits, where the number (the fraction) is stored in bits 0 to 51 , the exponent in bits 52 to 62, and the sign in bit 63.

Value	**Exponent**	**Sign**
52 bits (0 - 51)	11 bits (52 – 62)	1 bit (63)

Precision

Integers (numbers without a period or exponent notation) are accurate up to 15 digits.

E.g.

```
let x = 999999999999999;   // x will be 999999999999999
let y = 9999999999999999;  // y will be 10000000000000000
```

The maximum number of decimals is 17, but floating point arithmetic is not always 100% accurate.

Numeric Strings

JavaScript strings can have numeric content.

JavaScript will try to convert strings to numbers in all numeric operations.

NaN – Not a Number

NaN is a JavaScript reserved word indicating that a number is not a legal number.

Trying to do arithmetic with a non-numeric string will result in NaN (Not a Number).

User can use the global JavaScript function isNaN() to find out if a value is a not a number.

E.g.

```
let x = 100 / "Apple";  // x will be NaN (Not a Number)

isNaN(x);
```

Note : if user use NaN in a mathematical operation, the result will also be NaN.

NaN is number: typeof NaN returns number.

Infinity

Infinity (or –Infinity) is the value JavaScript will return if user calculate a number outside the largest possible number.

Division by 0 also generates Infinity.

Infinity is a number : typeof Infinity returns number.

Hexadecimal

JavaScript interprets numeric constant as hexadecimal if they are preceded by 0x.

E.g.

```
let x = 0xFF;        // x will be 255
```

By default, JavaScript displays numbers as base 10 decimals.

But user can use the toString() method to output numbers from base 2 to base 36.

Hexadecimal is base 16. Decimal is base 10. Octal is base 8. Binary is base2.

E.g.

```
let myNumber = 32;
myNumber.toString(10);  // returns 32
myNumber.toString(32);  // returns 10
myNumber.toString(16);  // returns 20
myNumber.toString(8);   // returns 40
myNumber.toString(2);   // returns 100000
```

JS Number Methods

The toString() Method

The toString() method returns a number as a string.

All number methods can be used on any type of number (literals, variables, or expressions)

The toExponential() Method

toExponential() returns a string, with a number rounded and written using exponential notation.

A parameter defines the number of characters behind the decimal point.

E.g.

```
let x = 9.656;
x.toExponential(2);    // returns 9.66e+0
x.toExponential(4);    // returns 9.6560e+0
```

The toFixed() Method

toFixed() returns a string, with the number written with a specified number of decimals.

E.g.

```
let x = 9.656;
x.toFixed(0);          // returns 10
x.toFixed(2);          // returns 9.66
x.toFixed(4);          // returns 9.6560
x.toFixed(6);          // returns 9.656000
```

The toPrecision() Method

toPrecision() returns a string, with a number written with a specified length.

E.g.

```
let x = 9.656;
x.toPrecision();       // returns 9.656
x.toPrecision(2);       // returns 9.7
x.toPrecision(4);       // returns 9.656
x.toPrecision(6);       // returns 9.65600
```

The valueOf() Method

valueOf() returns a number as a number.

E.g.

```
let x = 123;
x.valueOf();           // returns 123 from variable x
(123).valueOf();       // returns 123 from literal 123
(100 + 23).valueOf();   // returns 123 from
```

Note: All JavaScript data types have a valueOf() and a toString() method.

Converting Variables to Numbers

There are 3 JavaScript methods that can be used to convert variables to numbers:

- The Number() method
- The parseInt() method
- The parseFloat() method

These methods are not number methods, but global JavaScript methods.

E.g.

```
Number(true);        // returns 1
Number(false);       // returns 0
Number("10");        // returns 10
Number("  10");      // returns 10
Number("10  ");      // returns 10
Number(" 10  ");     // returns 10
Number("10.33");     // returns 10.33
Number("10,33");     // returns NaN
Number("John");      // returns NaN
```

Note: If the number cannot be converted, NaN (Not a Number) is returned.

E.g.

```
parseInt("-10");        // returns -10
parseInt("-10.33");     // returns -10
parseInt("10");         // returns 10
parseInt("10.33");      // returns 10
parseInt("10 20 30");   // returns 10
parseInt("10 years");   // returns 10
parseInt("years 10");   // returns NaN
```

E.g.

```
parseFloat("10");        // returns 10
parseFloat("10.33");     // returns 10.33
parseFloat("10 20 30");  // returns 10
parseFloat("10 years");  // returns 10
parseFloat("years 10");  // returns NaN
```

JS Conditions

Conditional statements are used to perform different actions based on different conditions.

Conditional Statements

Very often when user write code, user want to perform different actions for different decisions.

User can use conditional statements in our code to do this.

In JavaScript users have the following conditional statements:

- Use if to specify a block of code to be executed, if a specified condition is true.
- Use else to specify a block of code to be executed, if the same condition is false.
- Use else if to specify a new condition to test, if the first condition is false.
- Use switch to specify many alternative blocks of code to be executed.

The if Statement

Use the if statement to specify a block of JavaScript code to be executed if a condition is true.

Syntax

```
if (condition) {
  //  block of code to be executed if the condition is true
}
```

Note that if is in lowercase letters. Uppercase letters (If or IF) will generate a JavaScript error.

The else Statement

Use the else statement to specify a block of code to be executed if the condition is false.

```
if (condition) {
  //  block of code to be executed if the condition is true
} else {
  //  block of code to be executed if the condition is false
}
```

The else if Statement

Use the else if statement to specify a new condition if the first condition is false.

```
if (condition1) {
  //  block of code to be executed if condition1 is true
} else if (condition2) {
  //  block of code to be executed if the condition1 is false and condition2 is true
} else {
  //  block of code to be executed if the condition1 is false and condition2 is false
}
```

JS Switch

The switch statement is used to perform different actions based on different conditions.

The JavaScript Switch Statement

Use the switch statement to select one of many code blocks to be executed.

Syntax

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.
- If there is no match, the default code block is executed.

The break Keyword

When JavaScript reaches a break keyword, it breaks out of the switch block.
This will stop the execution inside the switch block.
It is not necessary to break the last case in a switch block. The block breaks (ends) there anyway.

The default Keyword

The default keyword specifies the code to run if there is no case match.

If default is not the last case in the switch block, remember to end the default case with a break.

Switching Details

If multiple cases matches a case value, the **first** case is selected.

If no matching cases are found, the program continues to the **default** label.

If no default label is found, the program continues to the statement(s) **after the switch**.

Strict Comparison

Switch cases use **strict** comparison (===).

The values must be of the same type to match.

A strict comparison can only be true if the operands are of the same type.

JS Loop For

Loops can execute a block of code a number of times.

Loops are handy, if users want to run the same code over and over again , each time with a different value.

Different Kinds of Loops

JavaScript supports 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 iteral 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.

The For Loop

The for loop has the following syntax:

```
for (initial ; condition ; increment or decrement ){
            // code block to be executed
}
```

E.g.

```
for (let i = 0; i < 5; i++) {
  console.log(`The number is ${i} `)
}
```

JS Loop For In

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

Syntax

```
for (key in object) {

  // code block to be executed

}
```

E.g.

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

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

Example Explained

- The for in loop iterates over a person object
- Each iteration return a key (x)
- The key is used to access the value of the key
- The value of the key is person[x]

JS Loop For of

The JavaScript for of statement loops through the values of an iterable object.

Syntax

```
for (variable of iterable) {
  // code block to be executed
}
```

variable - For every iteration the value of the next property is assigned to the variable. *Variable* can be declared with const, let, or var.

iterable - An object that has iterable properties.

Looping over an Array

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

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

Looping over a String

```
let language = "JavaScript";

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

JS Loop While

Loops can execute a block of code as long as a specified condition is true.

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

Syntax

```
while (condition) {

  // code block to be executed

}
```

E.g.

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

Note : If you forget to increase the variable used in the condition, the loop will never end. This will crash your browser.

The 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.

Syntax

```
do {

  // code block to be executed

}
while (condition);
```

E.g.

```
do {

  text += "The number is " + i;
  i++;

}
while (i < 10);
```

JS Arrays

JavaScript arrays are used to store multiple values in a single variable.

E.g.

const cars = [“Saab”, “Volvo”, “BMW”]

Note: It is a common practice to declare arrays with the const keyword.

What is an Array ?

An array is a special variable, which can hold more than one value at a time.

If user have a list of items (a list of car names, for example), storing the cars in single variable could look like this

```
let car1 = “Saab”;
let car2 = “Volvo”;
let car3 = “BMW”;
```

However, what if user want to loop through the cars and find a specific one? What if user had not 3 cars, but 300 ?

The solution is an array!

An array can hold many values under a single name, and user can access the values by referring to an index number.

Creating an Array

Using an array literal is the easiest way to create a JavaScript Array.

Syntax:

```
const array_name = [ item1 , item2, ……. ]
```

User can also create an array, and then provide the elements.

```
const cars = [];
cars[0] = “Saab”;
cars[1] = “Volvo”;
car[2] = “BMW”;
```

Using the JavaScript Keyword new

The following example also creates an Array, and assigns values to it:

```
const cars = new Array( “Saab”, “Volvo”, “BMW” );
```

The two examples above do exactly the same. There is no need to use new Array(). For simplicity, readability and execution speed, use the first one (the array literal method).

Accessing Array Elements

User access an array element by referring to the index number.

E.g.

```
const cars = ["Saab", "Volvo", "BMW"];
let x = cars[0];    // x = "Saab"
```

Note: Array indexes start with 0.

[0] is the first element. [1] is the second element.

Changing an Array Element

This statement changes the value of the first element in cars:

E.g.

```
const cars = ["Saab", "Volvo", "BMW"];
cars[0] = “Opel”;
```

Access the Full Array

With JavaScript, the full array can be accessed by referring to the array name:

```
const cars = ["Saab", "Volvo", "BMW"];
console.log(cars)
```

Arrays are Objects

Arrays are special type of objects. The typeof operator in JavaScript returns “object” for arrays.

But, JavaScript arrays are best described as arrays.

Arrays use numbers to access its “elements”. In this example, person[0] returns John.

Array Elements Can Be Objects

JavaScript variables can be objects. Arrays are special kinds of objects.

Because of this, user can have variables of different types in the same Array.

Array Properties and Methods

The real strength of JavaScript arrays are the built-in array properties and methods.

The length Property

The length property of an array returns the length of an array (the number of array elements).

E.g.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.length;   // Returns 4
```

Accessing the First Array Element

E.g.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[0];    // Returns "Banana"
```

Accessing the Last Array Element

E.g.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[fruits.length - 1];  // Returns "Mango"
```

Looping Array Elements

The safest way to loop throught an array, is using a for loop :

E.g.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fLen = fruits.length;

text = "<ul>";
for (let i = 0; i < fLen; i++) {
  text += "<li>" + fruits[i] + "</li>";
}
text += "</ul>";
```

User can also use the Array.forEach() function.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];

let text = "<ul>";
fruits.forEach(myFunction);
text += "</ul>";

function myFunction(value) {
  text += "<li>" + value + "</li>";
}
```

Adding Array Elements

The easiest way to add a new element to an array is using the push() method.

E.g.

```
const fruits = ["Banana", "Orange", "Apple"];
fruits.push("Lemon");  // Adds a new element (Lemon) to fruits
```

New element can also be added to an array using the length property

E.g.

```
const fruits = ["Banana", "Orange", "Apple"];
fruits[fruits.length] = "Lemon";  // Adds "Lemon" to fruits
```

Note : Adding elements with hight indexs can create undefined “holes” in an array.

The Difference Between Arrays and Objects

In JavaScript, arrays use numbered indexes.

In JavaScript, objects use named indexes.

Note: Arrays are a special kind of objects, with numbered indexes.

When to Use Arrays. When to use Objects.

- JavaScript does not support associative arrays.
- User should use objects when user want the element names to be string (text).
- User should use arrays when you want the element names to be numbers.

JavaScript new Array()

JavaScript has a built in array constructor new Array().

But user can safely use [] instead.

These two different statements both create a new empty array named points.

```
const points = new Array();
const points = [];
```

A Common Error

const points = [40]; is not the same as : const points = new Array (40);

```
// Create an array with one element:
const points = [40];
```

```
// Create an array with 40 undefined elements:
const points = new Array(40);
```

How to Recognize an Array

A common question is : How do I know if a variable is an array?

To solve this problem ECMAScript 5 defines a new method Array.isArray().

E.g.

```
const fruits = ["Banana", "Orange", "Apple"];

Array.isArray(fruits);   // returns true
```

JS Array Methods

Converting Arrays to Strings

The JavaScript method toString() converts an array to a string of (comma separated) array values .

The join() method also joins all array elements into a string.
It behaves just like toString(), but in addition use can specify the separators.

Popping and Pushing

When user work with arrays, it is easy to remove elements and add new elements.

This is what popping and pushing is : Popping items **out** of an array, or pushing items **into** an array.

Popping

1. The pop() method removes the last element from an array.
2. The pop() method returns the value that was “popped out”.

Pushing

1. The push() method adds a new element to an array (at the end).
2. The push() method returns the new array length.

Shifting Elements

Shifting is equivalent to popping, working on the first element instead of the last.

The shift() method removes the first array element and “shifts” all other elements to a lower index.

The shift() method returns the value that was “shifted out”.

The unshift() method adds a new element to an array (at the beginning), and "unshifts" older elements.

The unshift() method returns the new array length.

Splicing an Array

The splice() method can be used to add new items to an array.

E.g.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 0, "Lemon", "Kiwi");
```

The first parameter (2) defines the position where new element should be added.
The second parameter (0) defines how many element should be removed.
The rest of the parameters ("Lemon", "Kiwi") define the new elements to be added.
The splice() method returns an array with the deleted items.

Using splice() to Remove Elements

With clever parameter setting, user can use splice() to remove elements without leaving “holes” in the array.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(0, 1);
```

The first parameter (0) defines the position where new elements should be added (spliced in).

The second parameter (1) defines how many elements should be removed.

The rest of the parameters are omitted. No new elements will be added.

Merging (Concatenating) Arrays

The concat() method creates a new array by merging (concatenating) existing arrays.

E.g.

```
const myGirls = ["Cecilie", "Lone"];
const myBoys = ["Emil", "Tobias", "Linus"];

const myChildren = myGirls.concat(myBoys);
```

Note : The concat() method does not change the existing arrays. It always returns a new array.

The concat() method can take any number of array arguments.

E.g.

```
const arr1 = ["Cecilie", "Lone"];
const arr2 = ["Emil", "Tobias", "Linus"];
const arr3 = ["Robin", "Morgan"];
const myChildren = arr1.concat(arr2, arr3);
```

The concat() method can also take strings as argurments.

E.g.

```
const arr1 = ["Emil", "Tobias", "Linus"];
const myChildren = arr1.concat("Peter");
```

Slicing an Array

The slice() method slices out a piece of an array into a new array.

E.g.

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(1);
```

Note: The slice() method creates a new array. It does not remove any elements from the source array.

The slice() method can take two arguments like slice(1, 3).

The method then selects elements from the start argument, and up to (but not including) the end argument.

E.g.

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(1, 3);
```

JS Array Sort

Sorting an Array

The sort() method sorts an array alphabetically.

E.g.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
```

Reversing an Array

The reverse() method reverses the elements in an array.

E.g.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
fruits.reverse();
```

Numeric Sort

By default, the sort() function sorts values as strings.

This works well for strings (“Apple” come before “Banana”).

However, if numbers are sorted as strings, “25” is bigger than “100” , because “2” is bigger than “1”.

Because of this, the sort() method will produce incorrect result when sorting numbers.

User can fix this by providing a compare function.

E.g.

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b});
```

Find the Highest (or Lowest) Array Value

There are no built-in functions for finding the max or min value in an array.

However, after user have sorted an array, user can use the index to obtain the highest and lowest values.

E.g.

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b});
// now points[0] contains the lowest value
// and points[points.length-1] contains the highest value
```

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return b - a});
// now points[0] contains the highest value
// and points[points.length-1] contains the lowest value
```

Sorting Object Arrays

JavaScript arrays often contain objects

E.g.

```
const cars = [
  {type:"Volvo", year:2016},
  {type:"Saab", year:2001},
  {type:"BMW", year:2010}
];
```

Even if objects have properties of different data types, the sort() method can be used to sort the array.
The solution is to write a compare function to compare the property values:

```
cars.sort(function(a, b){return a.year - b.year});
```

JS Array Iteration

Array iteration methods operate on every array item.

Array.forEach()

The forEach() method calls a function (a callback function) once for each array element.

E.g.

```
const numbers = [45, 4, 9, 16, 25];
let txt = "";
numbers.forEach(myFunction);

function myFunction(value, index, array) {
  txt += value + "<br>";
}
```

Note : that the function takes 3 arguments:

- The item value
- The item index
- The array itself

Array.map()

The map() method creates a new array by performing a function on each array element.
The map() method does not execute the function for array elements without values.
The map() method does not change the original array.

E.g.

```
const numbers1 = [45, 4, 9, 16, 25];
const numbers2 = numbers1.map(myFunction);

function myFunction(value, index, array) {
  return value * 2;
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

Array.filter()

The filter() method creates a new array with array elements that passes a test.

This example creates a new array from elements with a value larger than 18.

E.g.

```
const numbers = [45, 4, 9, 16, 25];
const over18 = numbers.filter(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

Array.reduce()

The reduce() method runs a function on each array element to produce (reduce it to) a single value.

The reduce() method works from left-to-right in the array. See also reduceRight().

Note: The reduce() method does not reduce the original array.

E.g.

```
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduce(myFunction);

function myFunction(total, value, index, array) {
  return total + value;
}
```

Note : that the function takes 4 arguments:

- The total (the initial value / previously returned value)
- The item value
- The item index
- The array itself

Array.reduceRight()

The reduceRight() method runs a function on each array element to produce (reduce it to) a single value.

The reduceRight() works from right-to-left in the array.

E.g.

```
const numbers = [45, 4, 9, 16, 25];
let sum = numbers1.reduceRight(myFunction);

function myFunction(total, value, index, array) {
  return total + value;
}
```

Array.every()

The every() method check if all array values pass a test.

This example check if all array values are larger than 18:

E.g.

```
const numbers = [45, 4, 9, 16, 25];
let allOver18 = numbers.every(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

Array.some()

The some() method check if some array values pass a test.

This example check if some array values are larger than 18:

E.g.

```
const numbers = [45, 4, 9, 16, 25];
let someOver18 = numbers.some(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

Array.indexOf()

The indexOf() method searches an array for an element value and returns its position.

E.g.

```
const fruits = ["Apple", "Orange", "Apple", "Mango"];

let position = fruits.indexOf("Apple") + 1;
```

Note: Array.indexOf() returns -1 if the item is not found.

Array.lastIndexOf()

Array.lastIndexOf() is the same as Array.indexOf(), but returns the position of the last occurrence of the specified element.

E.g.

```
const fruits = ["Apple", "Orange", "Apple", "Mango"];
let position = fruits.lastIndexOf("Apple") + 1;
```

Array.inclues()

ECMAScript 2016 introduced Array.includes() to arrays. This allows us to check if an element is present in an array (including NaN, unlike indexOf).

E.g.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.includes("Mango"); // is true
```

Note: Array.inclues() allows to check for NaN values. Unlike Array.indexOf().

Array.find()

The find() method returns the value of the first array element that passes a test function.
This example finds (returns the value of) the first element that is larger than 18:

E.g.

```
const numbers = [4, 9, 16, 25, 29];
let first = numbers.find(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

Array.findIndex()

The findIndex() method returns the index of the first array element that passes a test function.
This example finds the index of the first element that is larger than 18:

E.g.

```
const numbers = [4, 9, 16, 25, 29];
let first = numbers.findIndex(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

JS Array Const

Arrays are Not Constants

The keyword const is a little misleading.

It does Not define a constant array. It defines a constant reference to an array.

Because of this, user can still change the elements of a constant array.

Elements Can be Reassigned

User can change the elements of a constant array.

E.g.

```
// You can create a constant array:
const cars = ["Saab", "Volvo", "BMW"];

// You can change an element:
cars[0] = "Toyota";

// You can add an element:
cars.push("Audi");
```

JS Dates

JavaScript Date Output

By default, JavaScript will use the browser’s time zone and display a date as a full text string.

E.g.

Sat Nov 13 2021 00:40:10 GMT+0630 (Myanmar Time)

Creating Date Objects

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

There are 4 ways to create a new date object.

- new Date()
- new Date(year, month, day, hours, minutes, seconds, milliseconds)
- new Date(milliseconds)
- new Date(date string)

new Date()

new Date() creates a new date object with the current date and time.

E.g.

```
const d = new Date();
```

Note : Date objects are static. The computer time is ticking, but date objects are not.

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

new Date(year, month, day, hours, minutes, seconds, milliseconds) creates a new date object with a specified date and time.

7 numbers specify year, month, day, hour, minute, second and millisecond (in that order).

E.g.

```
const d = new Date(2018, 11, 24, 10, 33, 30, 0);
```

Note : JavaScript counts months from 0 to 11 :

January = 0
December = 11

Using 6, 4, 3, or 2 Numbers

6 numbers specify year, month, day, hour, minute, second

E.g.

```
const d = new Date(2018, 11, 24, 10, 33, 30);
```

5 numbers specify year, month, day, hour, and minute.

E.g.

```
const d = new Date(2018, 11, 24, 10, 33);
```

4 numbers specify year, month, day, and hour:

E.g.

```
const d = new Date(2018, 11, 24, 10);
```

3 numbers specify year, month, and day.

E.g.

```
const d = new Date(2018, 11, 24);
```

2 numbers specify year and month:

E.g.

```
const d = new Date(2018, 11);
```

Previous Century

One and two digit years will be interpreted as 19xx .

E.g.

```
const d = new Date(99, 11, 24);
const d = new Date(9, 11, 24);
```

new Date (dateString)

new Date(dateString) creates a new date object from a **date string .**

E.g.

```
const d = new Date("October 13, 2014 11:13:00");
```

JavaScript Stores Dates as Milliseconds

JavaScript stores dates as number of milliseconds since January 01, 1970, 00:00:00 UTC (Universal Time Coordinated).

Note: Zero time is January 01, 1970 00:00:00 UTC.

new Date (milliseconds)

new Date(*milliseconds*) creates a new date object as **zero time plus milliseconds.**

E.g.

```
const d = new Date(0);
const d = new Date(100000000000);
const d = new Date(-100000000000);
```

Note : One day (24 hours) is 86 400 000 milliseconds.

Date Methods

When a Date object is created, a number of **methods** allow you to operate on it.

Date methods allow you to get and set the year, month, day, hour, minute, second, and millisecond of date objects, using either local time or UTC (Universal, or GMT) time.

Displaying Dates

JavaScript will (by default) output dates in full text string format.

When user display a date object in HTML , it is automatically converted to a string, with the toString() method.

E.g.

```
const d = new Date();
d.toString();
```

The toUTCString() method converts a date to a UTC string (a date display standard).

E.g.

```
const d = new Date();
d.toUTCString();
```

The toDateString() method converts a date to a more readable format.

E.g.

```
const d = new Date();
d.toDateString();
```

The toISOString() method converts a Date object to a string, using the ISO standard format.

E.g.

```
const d = new Date();
d.toISOString();
```

JS Date Formats

JavaScript Date Input

There are generally 3 types of JavaScript date input formats.

Type	Example
ISO Date	“2015-03-25” (The International Standard)
Short Date	“03/25/2015”
Long Date	“Mar 25 2015” or “25 Mar 2015”

Note : UTC (Universal Time Coordinated) is the same as GMT (Greenwich Mean Time).

JS Math

The JavaScript Math object allows user to perform mathematical tasks on numbers.

The Math Object

Unlike other objects, the Math object has no constructor.

The Math object is static.

All methods and properties can be used without creating a Math object first.

Math Properties (Constants)

The syntax for any Math properties is : Math.property .

JavaScript provides 8 mathematical constants that can be accessed as Math properties.

E.g.

```
Math.E        // returns Euler's number
Math.PI       // returns PI
Math.SQRT2    // returns the square root of 2
Math.SQRT1_2  // returns the square root of 1/2
Math.LN2      // returns the natural logarithm of 2
Math.LN10     // returns the natural logarithm of 10
Math.LOG2E    // returns base 2 logarithm of E
Math.LOG10E   // returns base 10 logarithm of E
```

Math Methods

The syntax for Math any method is : Math.method(number)

Number to Integer

There are 4 common methods to round a number to an integer.

Math.round(x)	Return x rounded to its nearest integer
Math.ceil(x)	Return x rounded up to its nearest integer
Math.floor(x)	Return x rounded down to its nearest integer
Math.trunc(x)	Returns the integer part of x (new in ES6)

Math.round()

Math.round(x) returns the nearest integer.

E.g.

```
Math.round ( 4.6 );
Math.round ( 4.5 );
Math.round (4.4 );
```

Math.ceil()

Math.ceil(x) returns the value of x rounded up to its nearest integer.

E.g.

```
Math.ceil(4.9);
Math.ceil(4.7);
Math.ceil(4.4);
Math.ceil(4.2);
Math.ceil(-4.2);
```

Math.floor()

Math.floor(x) returns the value of x rounded down to its nearest integer.

E.g.

```
Math.floor(4.9);
Math.floor(4.7);
Math.floor(4.4);
Math.floor(4.2);
Math.floor(-4.2);
```

Math.trunc()

Math.trunc(x) returns the integer part of x .

E.g.

```
Math.trunc(4.7);
Math.trunc(-4.4);
```

Math.sign()

Math.sign(x) returns if x is negative, null or positive.

E.g.

```
Math.sign(-4);
Math.sign(0);
Math.sign(4);
```

Math.pow()

Math.pow(x, y) returns the value of x to the power of y.

E.g.

```
Math.pow(8, 2);
```

Math.sqrt()

Math.sqrt(x) returns the square root of x .

E.g.

Math.sqrt(64);

Math.abs()

Math.abs(x) returns the absolute (positive) value of x.

E.g.

Math.abs(-4.7)

Math.sin( )

Math.sin(x) returns the sine (a value between -1 and 1) of the angle x (given in radians).

If user want to use degrees instead of radians, user have to convert degrees to radians.

Angle in radians = Angle in degrees x PI / 180.

E.g.

```
Math.sin(90 * Math.PI / 180);     // returns 1 (the sine of 90 degrees)
```

Math.cos()

Math.cos(x) returns the cosine (a value between -1 and 1) of the angle x (given in radians).

If user want to use degrees instead of radians , user have to convert degrees to radians.

Angle in radians = Angle in degrees x PI / 180.

E.g.

```
Math.cos(0 * Math.PI / 180);    // returns 1 (the cos of 0 degrees)
```

Math.min( ) and Math.max ( )

Math.min() and Math.max() can be used to find the lowest or highest value in a list of arguments.

E.g.

```
Math.min(0, 150, 30, 20, -8, -200);
Math.max(0, 150, 30, 20, -8, -200);
```

Math.random( )

Math.random( ) returns a random number between 0 (inclusive) and 1 (exclusive).

E.g.

```
Math.random( );
```

The Math.log( ) Method

Math.log(x) returns the natural logarithm of x.

The natural logarithm returns the time needed to reach a certain level of growth.

Math.E and Math.log() are twins.

E.g.

```
Math.log( 1 );
Math.log( 2 );
```

The Math.log2( ) Method

Math.log2(x) returns the base 2 logarithm of x .

E.g.

Math.log2(8)

The Math.log10( ) Method

Math.log10(x) returns the base 10 logarithm of x .

E.g.

Math.log10(1000);

JS Booleans

A JavaScript Boolean represents one of two values: true of false.

Boolean Values

Very often, in programming, user will need a data type that can only have one of two values, like

- Yes / No
- ON / OFF
- TRUE / FALSE

For this, JavaScript has a Boolean data type. It can only take the values true of false.

The Boolean( ) Function

Use can use the Boolean( ) function to find out if an expression (or a variable) is true .

E.g.

Boolean(10 > 9)

Note :

let x = false;
let y = new Boolean(false);

console.log(x == y) return true ;
console.log(x === y) return false ;

typeof x => Boolean
typeof y => Object

JS Comparisons

Conditional (Ternary) Operator

JavaScript also contains a conditional operator that assigns a value to a variable based on some condition.

Syntax:

```
variablename = ( condition ) ? value1 : value2 ;
```

JS Type of and Type Conversion

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

2 data types that cannot contain values :

- null
- undefined

The constructor Property

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

E.g.

```
"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]}
```

Difference Between Undefined and Null

Undefined and null are equal in value but different in type :

E.g.

```
typeof undefined          // undefined
typeof null               // object

null === undefined        // false
null == undefined         // true
```

Type Conversion

- Converting Strings to Numbers
- Converting Numbers to Strings
- Converting Numbers to Dates
- Converting Booleans to Numbers
- Converting Numbers to Booleans

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

Converting Strings to Numbers

The global method Number ( ) can convert strings to numbers.

Strings containing numbers (like "3.14") convert to numbers (like 3.14).

Empty strings convert to 0.

Anything else converts to NaN (Not a Number).

E.g.

```
Number("3.14")    // returns 3.14
Number(" ")       // returns 0
Number("")        // returns 0
Number("99 88")   // returns NaN
```

The Unary + Operator

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

E.g.

```
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):

E.g.

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

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:

E.g.

```
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
```

The Number method toString() does the same.

E.g.

```
x.toString()
(123).toString()
(100 + 23).toString()
```

Converting Dates to Numbers

The global method Number() can be used to convert dates to numbers.

E.g.

```
d = new Date();
Number(d)          // returns 1404568027739
```

The date method getTime() does the same.

E.g.

```
d = new Date();
d.getTime()        // returns 1404568027739
```

Converting Dates to Strings

The global method String() can convert dates to strings.

E.g.

```
String(Date())  // returns "Thu Jul 17 2014 15:38:19 GMT+0200 (W. Europe Daylight Time)"
```

The Date method toString() does the same.

E.g.

```
Date().toString()  // returns "Thu Jul 17 2014 15:38:19 GMT+0200 (W. Europe Daylight Time)"
```

Converting Booleans to Numbers

The global method Number() can also convert booleans to numbers.

E.g.

```
Number(false)     // returns 0
Number(true)      // returns 1
```

Converting Booleans to Strings

The global method String() can convert booleans to strings.

E.g.

```
String(false)      // returns "false"
String(true)       // returns "true"
```

JS this Keyword & Arrow Function

What is **this**?

The JavaScript this keyword refers to the object it belongs to .

It has different values depending on where it is used :

- In a method, this refers to the owner 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.
- Method like call() and apply() can refer this to any object.

this is a Method

In an object method, this refers to the “owner” of the method.

In the example on the top of this page, this refers to the person object.

The person object is the owner of the fullName method.

E.g.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  id: 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

this Alone

When used alone, the owner is the Global object, so this refers to the Global object.

In a browser window the Global object is [object Window] :

E.g.

```
let x = this ;
```

In strict mode, when used alone, this also refers to the Global object [object Window] :

E.g.

```
“use strict”;
Let x = this;
```

this is a Function (Default)

In a JavaScript function, the owner of the function is the **default** binding for this.

So, in a function, this refers to the Global object [object Window].

E.g.

```
function myFunction() {
  return this;
}
```

this in a Function (Strict)

JavaScript **strict mode** does not allow default binding.

So, when used in a function, in strict mode, this is undefined.

E.g.

```
"use strict";
function myFunction() {
  return this;
}
```

this in Event Handlers

In HTML event handlers, this refers to the HTML element that received the event :

E.g.

```
<button onclick="this.style.display='none'">
  Click to Remove Me!
</button>
```

Object Method Binding

In these examples, this is the **person** object (The person object is the "owner" of the function):

E.g.

```
const person = {
  firstName  : "John",
  lastName   : "Doe",
  id         : 5566,
  myFunction : function() {
    return this;
  }
};
```

JavaScript Arrow Function

Arrow functions were introduced in ES6.

Arrow functions allow us to write shorter function syntax:

E.g.

```
Let myFunction = (a, b) => a * b ;
```

E.g.

```
Hello = function ( ) {

            return “Hello World!“;
}
```

E.g.

```
hello = () => {

  return "Hello World!";

}
```

It gets shorter! If the function has only one statement, and the statement returns a value, you can remove the brackets *and* the return keyword:

```
hello = () => "Hello World!” ;
```

Note: This works only if the function has only one statement.

If users have parameters, user pass them inside the parentheses:

E.g.

```
hello = (val) => "Hello " + val;
```

In fact, if users have only one parameter, you can skip the parentheses as well:

E.g.

```
hello = val => "Hello " + val;
```