Introduction to JavaScript

Def

JavaScript is a high-level, *interpreted* programming language.

It was initially designed to run in web browsers to *make web pages interactive*.

It is a versatile language used for both *client-side and server-side development*.

JavaScript Reference Page → [Js Reference Page]()

Uses of Javascript

JavaScript Can Change HTML Content

```
document.getElementById("demo").innerHTML = "Hello JavaScript";
```

JavaScript Can Change HTML Attribute Values

```
document.getElementById(“myImage”).src=”pic_bulbon.gif”
```

JavaScript Can Change HTML Styles (CSS)

```
document.getElementById("demo").style.fontSize = "35px";
```

JavaScript Can Hide & Show HTML Elements

```
document.getElementById("demo").style.display = "none";
```

Used in (head, body, external)

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

```
<script>
  document.getElementById("demo").innerHTML = "My First JavaScript";
</script>
```

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

You can place an external script reference in <head> or <body> as you like.

To use an external script, Put the name of file in src attribute of <script> tag:

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

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

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

Js Output

Def

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().

1. innerHtml

To access an HTML element, JavaScript can use the document.getElementById(id) method.

The id attribute defines the HTML element.

The innerHTML property defines the HTML content:

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

2. document.write()

For testing purposes, it is convenient to use document.write():

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

Using document.write() after an HTML document is loaded, will **delete all existing HTML**:

3. window.alert()

You can use an alert box to display data:

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

In JavaScript, the window object is the global scope object.

This means that variables, properties, and methods by default belong to the window object.

This also means that specifying the *window keyword is optional*:

4. console.log()

For debugging purposes, call the console.log() method in the browser to display data.

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

5. window.print()

```
<button onclick="window.print()">Print this page</button>
```

Js Statements

Def

JavaScript statements are composed of:

Values, Operators, Expressions, Keywords, and Comments.

The statements are executed, one by one, in the same order as they are written.

Semicolons separate JavaScript statements.

```
let a, b, c;  // Declare 3 variables
a = 5;        // Assign the value 5 to a
b = 6;        // Assign the value 6 to b
c = a + b;    // Assign the sum of a and b to c
```

When separated by semicolons, multiple statements on one line are allowed:

```
a = 5; b = 6; c = a + b;
```

Js Syntax

Def

JavaScript syntax is the set of rules.

1. JavaScript Values

The JavaScript syntax defines two types of values:

- Fixed values
- Variable values

Fixed values are called **Literals**.

Variable values are called **Variables**.

Js 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:

Js Variables

In a programming language, **variables** are used to **store** data values.

JavaScript uses the keywords var, let and const to **declare** variables.

An **equal sign** is used to **assign values** to variables.

```
let x;
x = 6;
```

2. Js operators

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

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

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

3. Js Expressions

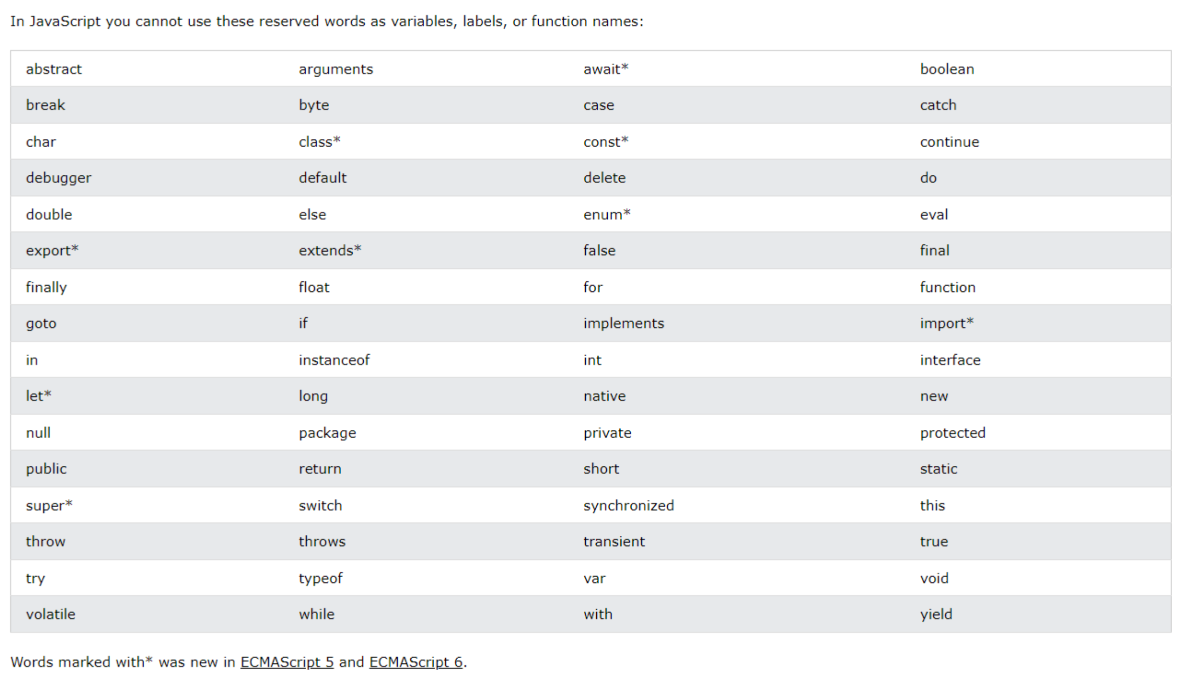
An expression is a combination of values, variables, and operators, which computes to value.

The computation is called an evaluation.

```
1.  5 * 10
2.  x * 10
3.  "John" + " " + "Doe"
```

4. Js Keywords

JavaScript **keywords** are used to identify actions to be performed.



In JavaScript you cannot use these reserved words as variables, labels, or function names:

abstract	arguments	await*	boolean
break	byte	case	catch
char	class*	const*	continue
debugger	default	delete	do
double	else	enum*	eval
export*	extends*	false	final
finally	float	for	function
goto	if	implements	import*
in	instanceof	int	interface
let*	long	native	new
null	package	private	protected
public	return	short	static
super*	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	yield

Words marked with* was new in ECMAScript 5 and ECMAScript 6.

5. Js Identifiers / Names

Identifiers are used to name variables and keywords, and functions.

A JavaScript name must begin with:

- A letter (A-Z or a-z)
- A dollar sign ($)
- Or an underscore (_)

Subsequent characters may be letters, digits, underscores, or dollar signs.

Numbers are not allowed as the first character in names.

6. Js Others

1. Js Comments

Code after double slashes // or between /* and */ is treated as a **comment**.

Single line comments start with //.

Multi-line comments start with /* and end with */.

2. Js is Case Sensitive

All JavaScript identifiers are **case sensitive**.

The variables lastName and lastname, are two different variables:

3. Js and CamelCase

JavaScript programmers tend to use camel case that starts with a lowercase letter:

firstName, lastName, masterCard, interCity.

4. Js Character Set

JavaScript uses the **Unicode** character set.

Unicode covers (almost) all the characters, punctuations, and symbols in the world.

Js Variables

Def

JavaScript Variables can be declared in 4 ways:

- Automatically
- Using var
- Using let
- Using const

The let and const keywords were added to JavaScript in 2015.

You cannot re-declare a variable declared with let or const.

When to use var, let and const

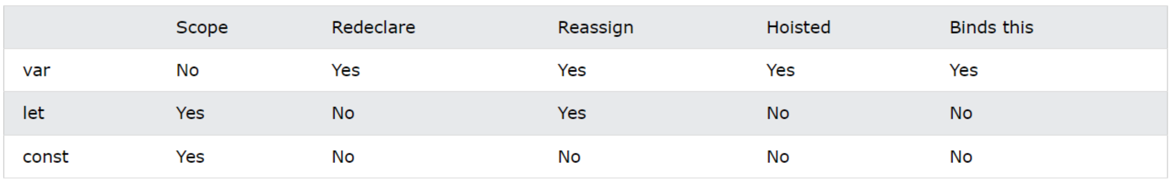
1. Always declare variables

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 and RegExp)

4. Only use let if you can't use const

5. Only use var if you MUST support old browsers.



	Scope	Redeclare	Reassign	Hoisted	Binds this
var	No	Yes	Yes	Yes	Yes
let	Yes	No	Yes	No	No
const	Yes	No	No	No	No

Hoisting

1. *Var Hoisting*

Variables defined with var are hoisted to the top and can be initialized at any time.

Meaning: You can use the variable before it is declared:

```
carName = "Volvo";

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

var carName;
```

2. *Let Hoisting*

Variables defined with let are also hoisted to the top of the block, but not initialized.

Meaning: Using a let variable before it is declared will result in a ReferenceError.

```
try {

  carName = "Saab";

  let carName = "Volvo";

}

catch(err) {

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

}
```

With **let**, you cannot use a variable before it is declared.

ReferenceError: Cannot access 'carName' before initialization

1. *Let*

Variables declared with let have **Block Scope**

Variables declared with let must be **Declared** before use

Variables declared with let cannot be **Redeclared** in the same scope

2. *Const*

Variables defined with const cannot be **Redeclared**

Variables defined with const cannot be **Reassigned**

Variables defined with const have **Block Scope**

JavaScript const variables *must be assigned* a value when they are declared:

Const Arrays

You can change the elements of a constant array:

But you can NOT reassign the array:

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

cars.push("Audi");  //Allowed
cars = ["Toyota", "Volvo", "Audi"];    // ERROR
```

Const Objects

You can change the properties of a constant object:

```
const car = {type:"Fiat", model:"500", color:"white"};

car.color = "red"; //Allowed

car = {type:"Volvo", model:"EX60", color:"red"};    // ERROR
```

Js Operators

Def

In JavaScript, an operator is a special symbol used to perform operations on operands (values and variables).

For example, 2 + 3; // 5. Here + is an operator that performs addition, and 2 and 3 are operands.

Types of JavaScript Operators

There are different types of JavaScript operators:

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- String Operators
- Logical Operators
- Bitwise Operators
- Ternary Operators
- Type Operators

1. *Arithmetic*

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

2. *Assignment*

i. *Arithmetic*

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

ii. *Logical*

Operator	Example	Same As
&&=	x &&= y	x = x && (x = y)
||=	x ||= y	x = x || (x = y)
??=	x ??= y	x = x ?? (x = y)

iii. *Shift*

Operator	Example	Same As
<<=	x <<= y	x = x << y
>>=	x >>= y	x = x >> y
>>>=	x >>>= y	x = x >>> y

iv. *Bitwise*

Operator	Example	Same As
&=	x &= y	x = x & y
^=	x ^= y	x = x ^ y
|=	x |= y	x = x | y

3. *Logical*

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

4. *Comparison*

Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

5. *Type*

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

6. *Bitwise*

Bit operators work on 32 bits numbers.

Any numeric operand in the operation is converted into a 32 bit number. The result is converted back to a JavaScript number.

Operator	Description	Example	Same as	Result	Decimal
&	AND	5 & 1	0101 & 0001	0001	1
|	OR	5 | 1	0101 | 0001	0101	5
~	NOT	~ 5	~0101	1010	10
^	XOR	5 ^ 1	0101 ^ 0001	0100	4
<<	left shift	5 << 1	0101 << 1	1010	10
>>	right shift	5 >> 1	0101 >> 1	0010	2
>>>	unsigned right shift	5 >>> 1	0101 >>> 1	0010	2

Js Data Types

Def

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 this:

let x = 16 + "Volvo";

Does it make any sense to add "Volvo" to sixteen? Will it produce an error or will it produce a result?

JavaScript will treat the example above as:

let x = "16" + "Volvo";

JavaScript Has 8 DataTypes:

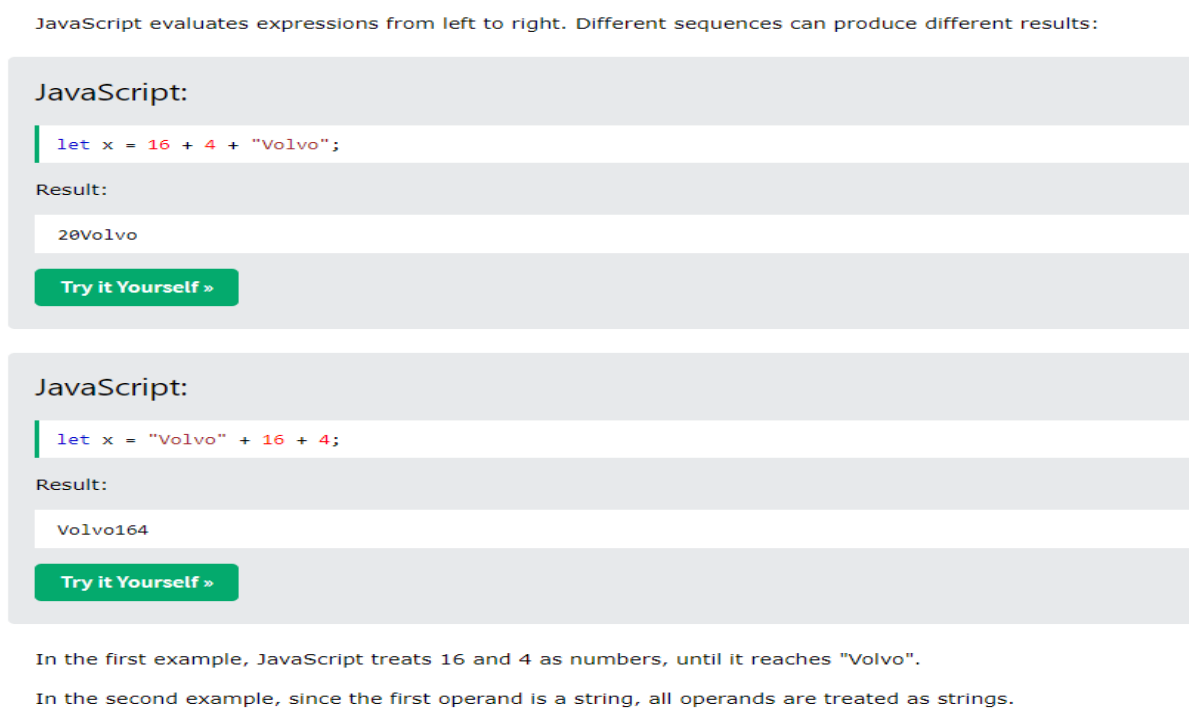
1. String
2. Number
3. Bigint
4. Boolean
5. Undefined
6. Null
7. Symbol
8. Object

The Object DataType:

1. An object
2. An array
3. A date

Examples

```
// Numbers:
let length = 16;
let weight = 7.5;

// Strings:
let color = "Yellow";
let lastName = "Johnson";

// Booleans
let x = true;
let y = false;

// Object:
const person = {firstName:"John", lastName:"Doe"};

// Array object:
const cars = ["Saab", "Volvo", "BMW"];

// Date object:
const date = new Date("2022-03-25");
```


JavaScript evaluates expressions from left to right. Different sequences can produce different results:

JavaScript:

```
let x = 16 + 4 + "Volvo";
```

Result:

```
20Volvo
```

JavaScript:

```
let x = "Volvo" + 16 + 4;
```

Result:

```
Volvo164
```

In the first example, JavaScript treats 16 and 4 as numbers, until it reaches "Volvo".

In the second example, since the first operand is a string, all operands are treated as strings.

Types of Data

1. ***Strings***

   A string (or a text string) is a series of characters like "John Doe".

   Strings are written with quotes. You can use single or double quotes

   You can use quotes inside a string, as long as they don't match quotes surrounding the string.

2. ***Numbers***

   All JavaScript numbers are stored as decimal numbers (floating point).

   Numbers can be written with, or without decimals:

   Extra large or extra small numbers can be written with scientific (exponential) notation:

   ```
   let y = 123e5;    // 12300000
   let z = 123e-5;   // 0.00123
   ```

3. **BigInt**

   All JavaScript numbers are stored in a a 64-bit floating-point format.

   JavaScript BigInt is a new datatype (ES2020) that can be used to store integer values that are too big to be represented by a normal JavaScript Number.

   ```
   let x = BigInt("123456789012345678901234567890");
   ```

4. ***Booleans***

   Booleans can only have two values: true or false.

5. ***Arrays***
JavaScript arrays are written with square brackets.

Array items are separated by commas.

The following code declares (creates) an array called cars, containing three items (car names):

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

6. ***Objects***
JavaScript objects are written with curly braces {}.

Object properties are written as name:value pairs, separated by commas.

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

7. ***Undefined***
In JavaScript, a variable without a value, has the value undefined. The type is also undefined.

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

Any variable can be emptied, by setting the value to undefined. The type will also be undefined.

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

8. ***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"
```

Js Functions

Def

A JavaScript function is a block of code designed to perform a particular task.

A JavaScript function is executed when "something" invokes it (calls it).

```
// Function to compute the product of p1 and p2
function myFunction(p1, p2) {
  return p1 * p2;
}
```

Syntax

```
function name(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

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

Function **parameters** are listed inside the parentheses () in the function definition.

Function **arguments** are the **values** received by the function when it is invoked.

Why Functions?

With functions you can reuse code

You can write code that can be used many times.

You can use the same code with different arguments, to produce different results.

Function Invocation

The code inside the function will execute when "something" **invokes** (calls) the function:

- When an event occurs (when a user clicks a button)
- When it is invoked (called) from JavaScript code
- Automatically (self invoked)

() Operator

The () operator invokes (calls) the function:

Convert Fahrenheit to Celsius:

```
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}
let value = toCelsius(77);
```

Accessing a function with incorrect parameters can return an incorrect answer:

Accessing a function without () returns the function and not the function result:

toCelsius refers to the function object, and *toCelsius()* refers to the function result

Js Objects

Def

In real life, a car is an **object**.

A car has **properties** like weight and color, and **methods** like start and stop

Object	**Properties**	**Methods**

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

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**.

Objects are variables too. But objects can contain many values.

This code assigns **many values** (Fiat, 500, white) to a **variable** named car

```
const car = {type:"Fiat", model:"500", color:"white"};
```

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.

Object Definition

You define (and create) a JavaScript object with an object literal:

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

Spaces and line breaks are not important. An object definition can span multiple lines:

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

Assessing Object Properties

You can access object properties in two ways:

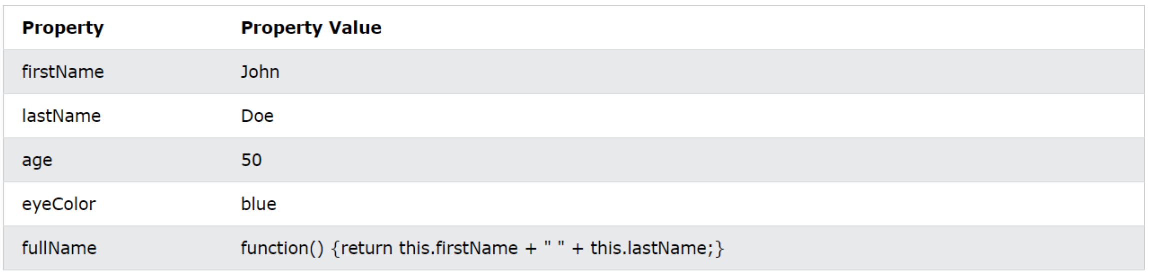
```
objectName.propertyName
```

OR

```
objectName["propertyName"]
```

Object Methods

Objects can also have **methods**.



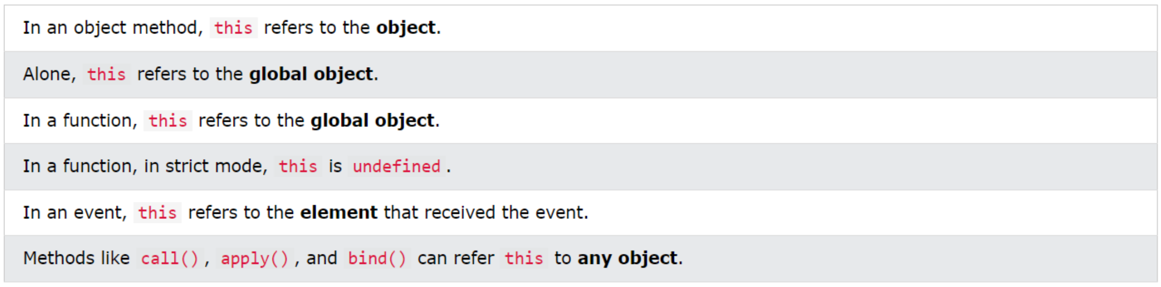
Property	**Property Value**
firstName	John
lastName	Doe
age	50
eyeColor	blue
fullName	function() {return this.firstName + " " + this.lastName;}

What is This?

In JavaScript, the *this* keyword refers to an **object**.

Which object depends on how this is being invoked (used or called).

The *this* keyword refers to different objects depending on how it is used



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**.

Do Not Declare Strings, Numbers, and Booleans as Objects!

When a JavaScript variable is declared with the keyword "new", the variable is created as an object:

```
x = new String();        // Declares x as a String object
y = new Number();        // Declares y as a Number object
z = new Boolean();       // Declares z as a Boolean object
```

Avoid String, Number, and Boolean objects. They complicate your code and slow down execution speed.

Js Events

Def

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

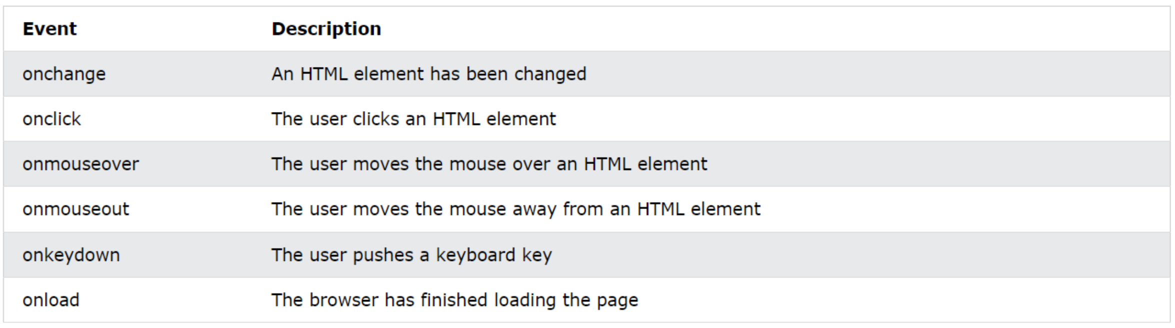
Here are some examples of HTML events:

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

HTML allows event handler attributes, with JavaScript code, to be added to HTML elements.

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

Comman 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 pushes a keyboard key
onload	The browser has finished loading the page

Js Strings

Def

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

```
let text = "John Doe";
```

Quotes Inside Quotes

You can use quotes inside a string, as long as they don't match the quotes surrounding the string:

```
let answer1 = "It's alright";
let answer2 = "He is called 'Johnny'";
let answer3 = 'He is called "Johnny"';
```

Template Strings

Templates are strings enclosed in backticks (`This is a template string`).

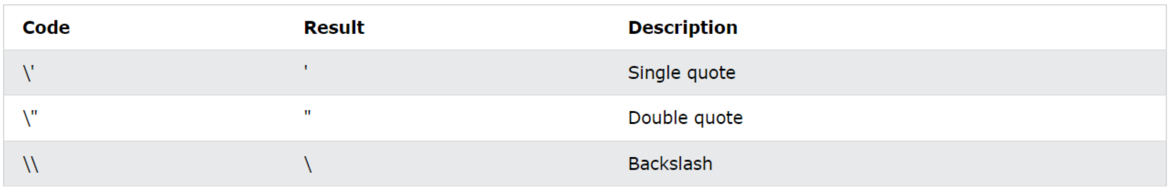
Templates allow single and double quotes inside a string:

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

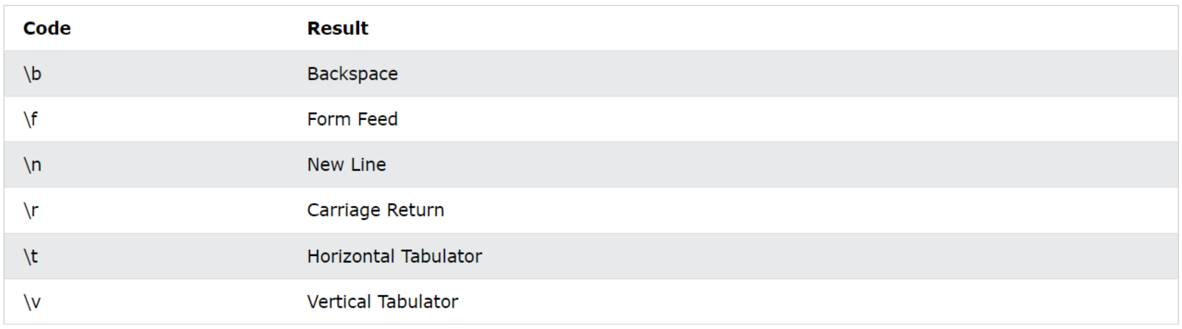
Escape Characters

you can use a **backslash escape character**.

The backslash escape character (\) turns special characters into string characters:



Code	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash



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

JavaScript Strings as Objects

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

let x = "John";
let y = new String("John");

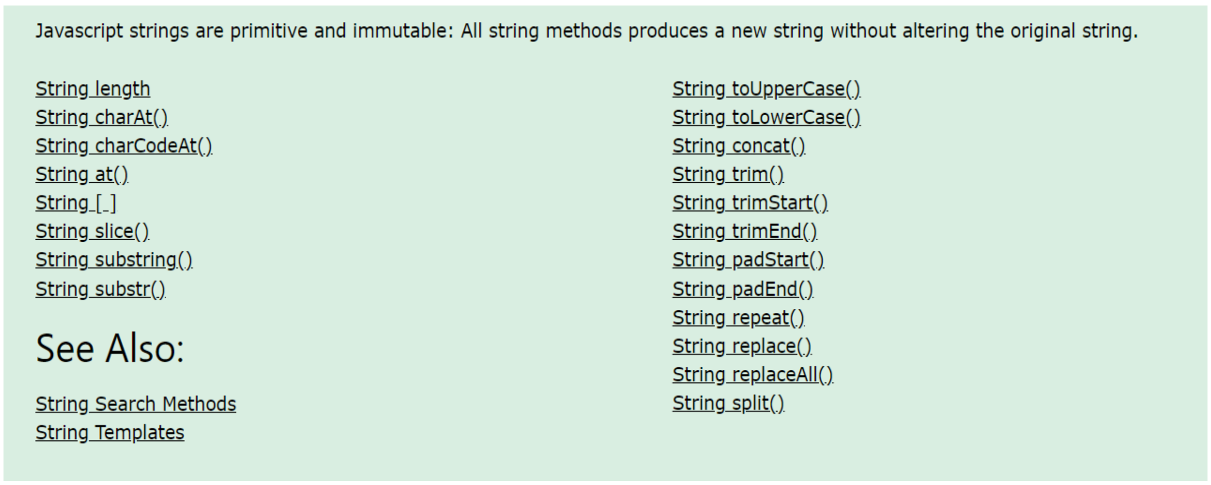
The new keyword complicates the code and slows down execution speed.

String objects can produce unexpected results:

difference between (x==y) and (x===y).

Basic String Methods

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



Javascript strings are primitive and immutable: All string methods produces a new string without altering the original string.

String length
String charAt()
String charCodeAt()
String at()
String [ ]
String slice()
String substring()
String substr()

See Also:

String Search Methods
String Templates

String toUpperCase()
String toLowerCase()
String concat()
String trim()
String trimStart()
String trimEnd()
String padStart()
String padEnd()
String repeat()
String replace()
String replaceAll()
String split()

Length

The length property returns the length of a string:

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

Extracting String Characters

There are 4 methods for extracting string characters:

- The *at(position)* Method
- The *charAt(position)* Method
- The *charCodeAt(position)* Method
- Using *property access []* like in arrays

The at() method is a new addition to JavaScript.

It allows the use of negative indexes while charAt() do not.

Extracting String Parts

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

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

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

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

substring() is similar to slice().

The difference is that start and end values less than 0 are treated as 0 in substring().

substr() is similar to slice().

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

Converting to Upper and Lower Case

A string is converted to upper case with toUpperCase()

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

String concat()

concat() joins two or more strings

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

Trim Methods

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

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

The trimStart() method works like trim(), but removes whitespace only from the start of a string.

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

The trimEnd() method works like trim(), but removes whitespace only from the end of a string.

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

Padding Methods

The padStart() method pads a string from the start.

It pads a string with another string (multiple times) until it reaches a given length.

```
let text = "5";
let padded = text.padStart(4,"0");
```

The padEnd() method pads a string from the end.

It pads a string with another string (multiple times) until it reaches a given length.

```
let text = "5";
let padded = text.padEnd(4,"0");
```

String* repeat*()

The repeat() method returns a string with a number of copies of a string.

string.repeat(count)

```
let text = "Hello world!";
let result = text.repeat(2);
```

replace Methods

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

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

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

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

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

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

The replaceAll() method allows you to specify a regular expression instead of a string to be replaced.

Converting a String to an Array

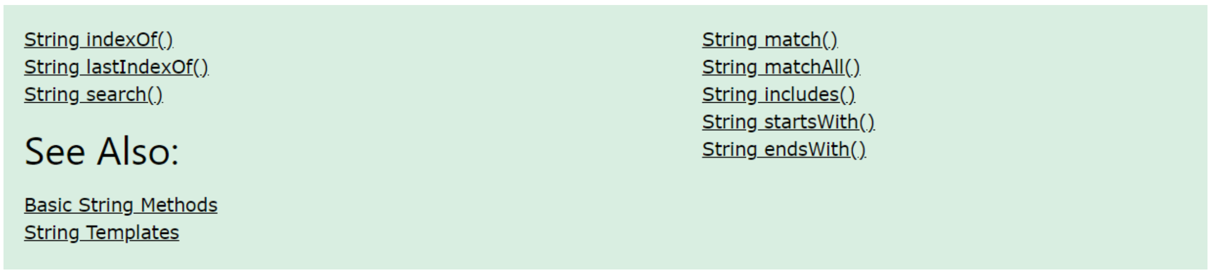
A string can be converted to an array with the split() method

```
text.split(",")    // Split on commas
text.split(" ")    // Split on spaces
text.split("|")    // Split on pipe
```

If the separator is omitted, the returned array will contain the whole string in index [0].

If the separator is "", the returned array will be an array of single characters:

String Search Methods



String indexOf()
String lastIndexOf()
String search()

See Also:

Basic String Methods
String Templates

String match()
String matchAll()
String includes()
String startsWith()
String endsWith()

return as index

The indexOf() method returns the **index** (position) of the **first** occurrence of a string in a string, or it returns -1 if the string is not found

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

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

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

Both methods accept a second parameter as the starting position for the search:

```
let text = "Please locate where 'locate' occurs!";
let index = text.indexOf("locate", 15);
```

The lastIndexOf() methods searches backwards (from the end to the beginning), meaning: if the second parameter is 15, the search starts at position 15, and searches to the beginning of the string.

The search() method searches a string for a string (or a regular expression) and returns the position of the match

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

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

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

return as array

The match() method returns an array containing the results of matching a string against a string (or a regular expression).

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

The matchAll() method returns an iterator containing the results of matching a string against a string (or a regular expression).

```
const iterator = text.matchAll("Cats");
```

return as boolean

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

```
let text = "Hello world, welcome to the universe.";
text.includes("world", 12);
```

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

```
let text = "Hello world, welcome to the universe.";
text.startsWith("world", 6)
```

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

Check if the 11 first characters of a string ends with "world"

```
let text = "Hello world, welcome to the universe.";
text.endsWith("world", 11);
```

String Interpolation

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

The method is called string interpolation.

The syntax is:

${...}

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

Template Strings allow expressions in strings:

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

HTML Templates

```
let header = "Template Strings";
let tags = ["template strings", "javascript", "es6"];
let html = `<h2>${header}</h2><ul>`;
for (const x of tags) { html += `<li>${x}</li>`;  }
html += `</ul>`;
```

Js Numbers

Def

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

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

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

```
let x = 123e5;    // 12300000
let y = 123e-5;   // 0.00123
```

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 numbers, following 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 (aka Fraction/Mantissa)	Exponent	Sign
52 bits (0 - 51)	11 bits (52 - 62)	1 bit (63)

Floating point arithmetic is not always 100% accurate:

0.2 + 0.1 = 0.30000000000000004

But it helps to multiply and divide:

0.2 + 0.1 = 0.3

```
let x = 0.2 + 0.1;

document.getElementById("demo1").innerHTML = "0.2 + 0.1 = " + x;

let y = (0.2*10 + 0.1*10) / 10;

document.getElementById("demo2").innerHTML = "0.2 + 0.1 = " + y;
```

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

The Code which works:

```
let x = "100";
let y = "10";
```

- *let z = x / y;*
- *let z = x * y;*
- *let z = x - y;*

The code which doesn't work:

```
let z = x + y;
```

NaN

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

```
let x = 100 / "Apple";
isNaN(x);
```

Infinity

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

```
let myNumber = 2;
// Execute until Infinity
while (myNumber != Infinity) {
  myNumber = myNumber * myNumber;
}
```

Number Objects

Normally JavaScript numbers are primitive values created from literals:

```
let x = 123;
```

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

```
let y = new Number(123);
```

BigInt

JavaScript BigInt variables are used to store big integer values that are too big to be represented by a normal JavaScript Number.

To create a BigInt, append n to the end of an integer or call BigInt():

```
let x = 12345678901234567890012345;

let y = 12345678901234567890012345n;

let z = BigInt(12345678901234567890012345)
```

JavaScript can only safely represent integers:

Up to **9007199254740991** +(2^{53}-1)

and

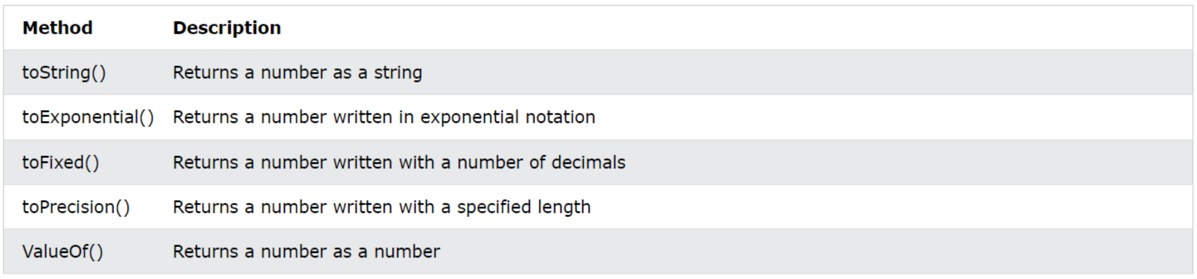
Down to **-9007199254740991** -(2^{53}-1).

Integer values outside this range lose precision.

```
Number.isInteger()

Number.isSafeInteger()
```

Number Methods

These **number methods** can be used on all JavaScript numbers:



Method	Description
toString()	Returns a number as a string
toExponential()	Returns a number written in exponential notation
toFixed()	Returns a number written with a number of decimals
toPrecision()	Returns a number written with a specified length
ValueOf()	Returns a number as a number

toString()

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

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

```
let x = 123;
x.toString();                123
(123).toString();            123
(100 + 23).toString();       123
```

toExponential()

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

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

toFixed()

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

```
let x = 9.656;
x.toFixed(0);          10
x.toFixed(3);          9.656
```

toPrecision()

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

```
let x = 9.656;
x.toPrecision();             9.656
x.toPrecision(3);            9.66
```

valueof()

valueOf() returns a number as a number.

The valueOf() method is used to convert Number objects to primitive values.

```
let x = 123;
x.valueOf();                 123
(123).valueOf();             123
(100 + 23).valueOf();        123
```

Converting Variables to Numbers

There are 3 JavaScript methods that can be used to convert a variable to a number:

Method	Description
Number()	Returns a number converted from its argument.
parseFloat()	Parses its argument and returns a floating point number
parseInt()	Parses its argument and returns a whole number

Number Object Methods

These **object methods** belong to the **Number** object:

Method	Description
Number.isInteger()	Returns true if the argument is an integer
Number.isSafeInteger()	Returns true if the argument is a safe integer
Number.parseFloat()	Converts a string to a number
Number.parseInt()	Converts a string to a whole number

Number Properties

Number properties belong to the JavaScript **Number Object**.

Property	Description
EPSILON	The difference between 1 and the smallest number > 1.
MAX_VALUE	The largest number possible in JavaScript
MIN_VALUE	The smallest number possible in JavaScript
MAX_SAFE_INTEGER	The maximum safe integer (2^{53} - 1)
MIN_SAFE_INTEGER	The minimum safe integer -(2^{53} - 1)
POSITIVE_INFINITY	Infinity (returned on overflow)
NEGATIVE_INFINITY	Negative infinity (returned on overflow)
NaN	A "Not-a-Number" value

EPSILON

Number.EPSILON is the difference between the smallest floating-point number greater than 1 and 1.

let x = Number.EPSILON; *2.220446049250313e-16*

MAX_VALUE

Number.MAX_VALUE is a constant representing the largest possible number in JavaScript.

let x = Number.MAX_VALUE; *1.7976931348623157e+308*

MIN_VALUE

Number.MIN_VALUE is a constant representing the lowest possible number in JavaScript.

let x = Number.MIN_VALUE;	*5e-324*

MAX_SAFE_INTEGER

Number.MAX_SAFE_INTEGER represents the maximum safe integer in JavaScript.

Number.MAX_SAFE_INTEGER is (2^{53} - 1).

let x = Number.MAX_SAFE_INTEGER;	*9007199254740991*

MIN_SAFE_INTEGER

Number.MIN_SAFE_INTEGER represents the minimum safe integer in JavaScript.

Number.MIN_SAFE_INTEGER is -(2^{53} - 1).

let x = Number.MIN_SAFE_INTEGER;	*-9007199254740991*

POSITIVE_INFINITY

let x = Number.POSITIVE_INFINITY;	*Infinity*

NEGATIVE_INFINITY

let x = Number.NEGATIVE_INFINITY;	*-Infinity*

NaN - Not a Number

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

let x = Number.NaN	*NaN*

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

```
let x = 100 / "Apple";
```

Js Arrays

Def

An array is a special variable, which can hold more than one value:

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

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

You access an array element by referring to the **index number**:

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

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

```
cars[0] = "Opel";
```

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

```
document.getElementById("demo").innerHTML = cars;
```

Arrays are Object

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

But JavaScript arrays are best described as arrays.

You can have objects in an Array. You can have functions in an Array. You can have arrays in an Array:

```
myArray[0] = Date.now;
myArray[1] = myFunction;
myArray[2] = myCars;
```

Looping Array Elements

One way to loop through an array, is using a for loop:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fLen = fruits.length;
let text = "<ul>";
for (let i = 0; i < fLen; i++) {
  text += "<li>" + fruits[i] + "</li>";
}
text += "</ul>";
```

You 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>";
}
```

Associative Arrays

Arrays with named indexes are called associative arrays (or hashes).

JavaScript does **not** support arrays with named indexes.

In JavaScript, **arrays** always use **numbered indexes**.

Array vs Objects

In JavaScript, **arrays** use **numbered indexes**.

In JavaScript, **objects** use **named indexes**.

- JavaScript does not support associative arrays.
- You should use **objects** when you want the element names to be **strings (text)**.
- You should use **arrays** when you want the element names to be **numbers**.

How to Recognize an Array

The problem is that the JavaScript operator typeof returns "object":

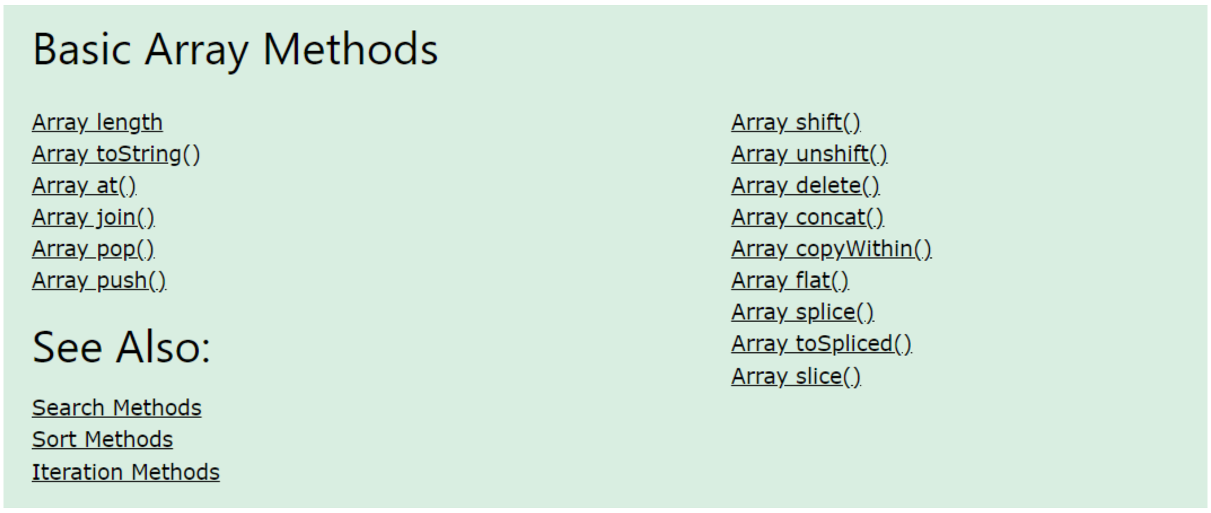
To solve this problem ECMAScript 5 (JavaScript 2009) defined a new method Array.isArray():

```
Array.isArray(fruits);
```

The instanceof operator returns true if an object is created by a given constructor:

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

Array Methods



Basic Array Methods

Array length
Array toString()
Array at()
Array join()
Array pop()
Array push()

Array shift()
Array unshift()
Array delete()
Array concat()
Array copyWithin()
Array flat()
Array splice()
Array toSpliced()
Array slice()

See Also:

Search Methods
Sort Methods
Iteration Methods

length

The length property returns the length (size) of an array:

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

toString()

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

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.toString();
```

- *Result:*

```
Banana,Orange,Apple,Mango
```

at()

Many languages allow negative bracket indexing like [-1] to access elements from the end of an object / array / string.

This is not possible in JavaScript, because [] is used for accessing both arrays and objects. obj[-1] refers to the value of key -1, not to the last property of the object.

The at() method was introduced in ES2022 to solve this problem.

The at() method returns an indexed element from an array.

The at() method returns the same as [].

```
let fruit = fruits.at(2);
```

join()

The join() method also joins all array elements into a string.

It behaves just like toString(), but in addition you can specify the separator:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.join(" * ");
```

```
Result:         Banana * Orange * Apple * Mango
```

Popping and Pushing

pop()

The pop() method removes the last element from an array:

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

The pop() method returns the value that was "popped out":

push()

The push() method adds a new element to an array (at the end):

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

The push() method returns the new array length:

Shifting Elements

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

shift()

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

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

The shift() method returns the value that was "shifted out":

unshift()

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

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

The unshift() method returns the new array length:

delete()

Using delete() leaves undefined holes in the array.

Use pop() or shift() instead.

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

Merging Arrays (Concatenating)

concat()

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

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

const myChildren = arr1.concat(arr2, arr3);
```

copywithin()

The copyWithin() method copies array elements to another position in an array:

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

Result:          Banana,Orange,Banana,Orange

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

Result:          Banana,Orange,Banana,Orange,Kiwi,Papaya
```

The copyWithin() method overwrites the existing values.

The copyWithin() method does not add items to the array.

The copyWithin() method does not change the length of the array.

Flattening an Array

Flattening an array is the process of reducing the dimensionality of an array.

Flattening is useful when you want to convert a multi-dimensional array into a one-dimensional array.

flat()

The flat() method creates a new array with sub-array elements concatenated to a specified depth.

```
const myArr = [[1,2],[3,4],[5,6]];
const newArr = myArr.flat();
```

Splicing and Slicing Arrays

The splice() method adds new items to an array.

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

splice()

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

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

The splice() method returns an array with the deleted items:

tospliced()

The difference between the new **toSpliced()** method and the old **splice()** method is that the new method creates a new array, keeping the original array unchanged, while the old method altered the original array.

const months = ["Jan", "Feb", "Mar", "Apr"];
const spliced = months.toSpliced(0, 1);

slice()

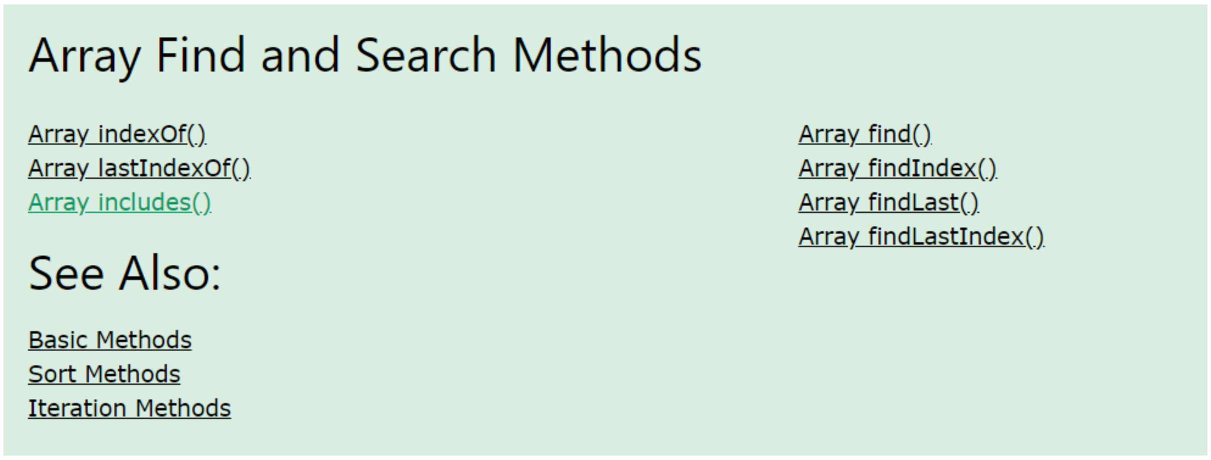
The slice() method slices out a piece of an array into a new array:

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

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.

Search Methods



Array Find and Search Methods

Array indexOf()
Array lastIndexOf()
Array includes()

Array find()
Array findIndex()
Array findLast()
Array findLastIndex()

See Also:

Basic Methods
Sort Methods
Iteration Methods

indexOf()

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

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

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

If the item is present more than once, it returns the position of the first occurrence.

lastIndexOf()

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

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

includes()

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

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

find()

The find() method returns the value of the first array element that passes a test function.

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


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

findIndex()

The findIndex() method returns the index of the first array element that passes a test function.

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


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

findLast()

ES2023 added the findLast() method that will start from the end of an array and return the value of the first element that satisfies a condition.

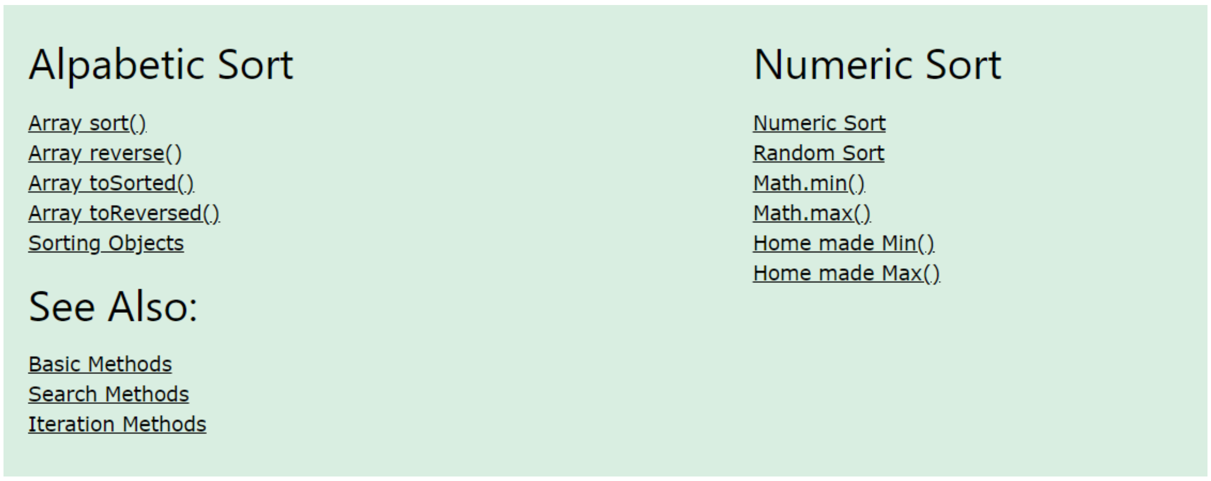
```
const temp = [27, 28, 30, 40, 42, 35, 30];
let high = temp.findLast(x => x > 40);
```

findLastIndex()

The findLastIndex() method finds the index of the last element that satisfies a condition.

```
let pos = temp.findLastIndex(x => x > 40);
```

Sorting Methods



Alpabetic Sort

Array sort()
Array reverse()
Array toSorted()
Array toReversed()
Sorting Objects

See Also:

Basic Methods
Search Methods
Iteration Methods

Numeric Sort

Numeric Sort
Random Sort
Math.min()
Math.max()
Home made Min()
Home made Max()

Sorting an Array

sort()

The sort() method sorts an array alphabetically:

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

reverse()

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

```
fruits.reverse();
```

By combining sort() and reverse(), you can sort an array in descending order:

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

toSorted()

The difference between toSorted() and sort() is that the first method creates a new array, keeping the original array unchanged, while the last method alters the original array.

```
const months = ["Jan", "Feb", "Mar", "Apr"];
const sorted = months.toSorted();
```

toReversed()

The difference between toReversed() and reverse() is that the first method creates a new array, keeping the original array unchanged, while the last method alters the original array.

```
const reversed = months.toReversed();
```

Numeric Sort

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

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.

You can fix this by providing a **compare function**:

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

Sorting in a Random Order

Using a sort function, like explained above, you can sort an numeric array in random order

```
points.sort(function(){return 0.5 - Math.random()});
```

The Fisher Yates Method

The points.sort() method in the example above is not accurate. It will favor some numbers over others.

The most popular correct method, is called the Fisher Yates shuffle, and was introduced in data science as early as 1938!

In JavaScript the method can be translated to this:

```
const points = [40, 100, 1, 5, 25, 10];

for (let i = points.length -1; i > 0; i--) {
  let j = Math.floor(Math.random() * (i+1));
  let k = points[i];
  points[i] = points[j];
  points[j] = k;
}
```

Find the Lowest (or Highest) Array Value

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

To find the lowest or highest valu you have 3 options:

- Sort the array and read the first or last element
- Use Math.min() or Math.max()
- Write a home-made function

Math.min.apply(null, [1, 2, 3]) is equivalent to Math.min(1, 2, 3).

Math.max.apply(null, [1, 2, 3]) is equivalent to Math.max(1, 2, 3).

Home-made Min Function

```
function myArrayMin(arr) {
  let len = arr.length;
  let min = Infinity;
  while (len--) {
    if (arr[len] < min) {
      min = arr[len];
    }
  }
  return min;
}
```

Home-made Max Function

```
function myArrayMax(arr) {
  let len = arr.length;
  let max = -Infinity;
  while (len--) {
    if (arr[len] > max) {
      max = arr[len];
    }
  }
  return max;
}
```

Soring Object Arrays

Even if objects have properties of different data types, the sort() method can be used to sort the array.

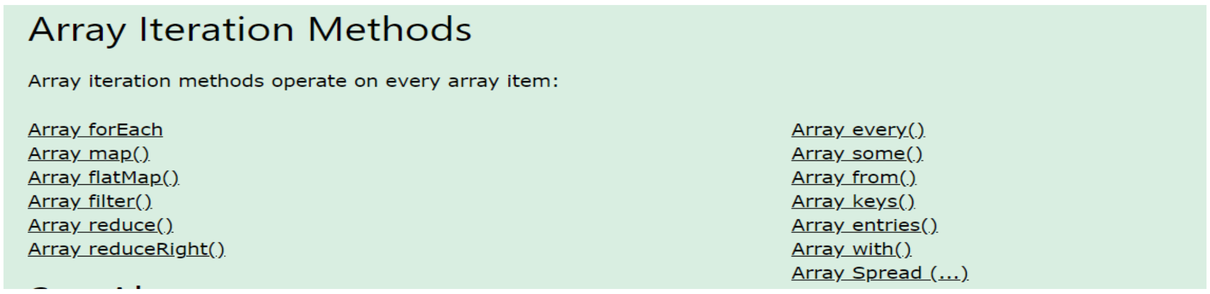
The solution is to write a compare function to compare the property values:

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

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

Comparing string properties is a little more complex:

```
cars.sort(function(a, b){
  let x = a.type.toLowerCase();
  let y = b.type.toLowerCase();
  if (x < y) {return -1;}
  if (x > y) {return 1;}
  return 0;
});
```

Iteration Methods



Array Iteration Methods

Array iteration methods operate on every array item:

Array forEach
Array map()
Array flatMap()
Array filter()
Array reduce()
Array reduceRight()

Array every()
Array some()
Array from()
Array keys()
Array entries()
Array with()
Array Spread (...)

forEach()

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

```
const numbers = [45, 4, 9, 16, 25];
let txt = "";
numbers.forEach(myFunction);
function myFunction(value, index, array) {
  txt += value + "<br>";
}
```

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.

```
const numbers2 = numbers1.map(myFunction);
function myFunction(value, index, array) {
  return value * 2;
}
```

flatMap()

The flatMap() method first maps all elements of an array and then creates a new array by flattening the array.

```
const myArr = [1, 2, 3, 4, 5, 6];
const newArr = myArr.flatMap((x) => x * 2);
```

filter()

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

```
const over18 = numbers.filter(myFunction);
function myFunction(value, index, array) {
  return value > 18;
}
```

reduce()

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

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

```
let sum = numbers.reduce(myFunction);
function myFunction(total, value, index, array) {
  return total + value;
}
```

The reduce() method can accept an initial value:

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. See also reduce().

```
let sum = numbers.reduceRight(myFunction);
function myFunction(total, value, index, array) {
  return total + value;
}
```

every()

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

```
let allOver18 = numbers.every(myFunction);
function myFunction(value, index, array) {
  return value > 18;
}
```

some()

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

```
let someOver18 = numbers.some(myFunction);
```

form()

The Array.from() method returns an Array object from any object with a length property or any iterable object.

```
Array.from("ABCDEFG");
```

Keys()

The Array.keys() method returns an Array Iterator object with the keys of an array.

```
const keys = fruits.keys();
```

entries()

Create an Array Iterator, and then iterate over the key/value pairs:

```
const f = fruits.entries();
```

with()

ES2023 added the Array with() method as a safe way to update elements in an array without altering the original array.

```
const months = ["Januar", "Februar", "Mar", "April"];
const myMonths = months.with(2, "March");
```

spread (...)

The ... operator expands an iterable (like an array) into more elements:

```
const q1 = ["Jan", "Feb", "Mar"];
const q2 = ["Apr", "May", "Jun"];
const q3 = ["Jul", "Aug", "Sep"];
const q4 = ["Oct", "Nov", "May"];

const year = [...q1, ...q2, ...q3, ...q4];
```

Array Const

In 2015, JavaScript introduced an important new keyword: const.

It has become a common practice to declare arrays using const:

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

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, we can still change the elements of a constant array.

You can change the elements of a constant array:

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

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

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

Redeclaring an array declared with var is allowed anywhere in a program:

Redeclaring or reassigning an array to const, in the same scope, or in the same block, is not allowed:

Redeclaring or reassigning an existing const array, in the same scope, or in the same block, is not allowed:

Redeclaring an array with const, in another scope, or in another block, is allowed:

Js Date (Not fully Covered)

Def

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)
```

Specifying month higher than 11, will not result in error but add overflow to the next year:

Date()

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

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

```
Result:  Thu Feb 15 2024 22:06:14 GMT+0530 (India Standard Time)
```

new Date(date string)

new Date(*date string*) creates a date object from a **date string**:

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

new Date(year, month, ...)

new Date(*year, month, ...*) creates a date object with a **specified date and time**.

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

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

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

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

JavaScript Stores Dates as Milliseconds

JavaScript stores dates as number of milliseconds since January 01, 1970.

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

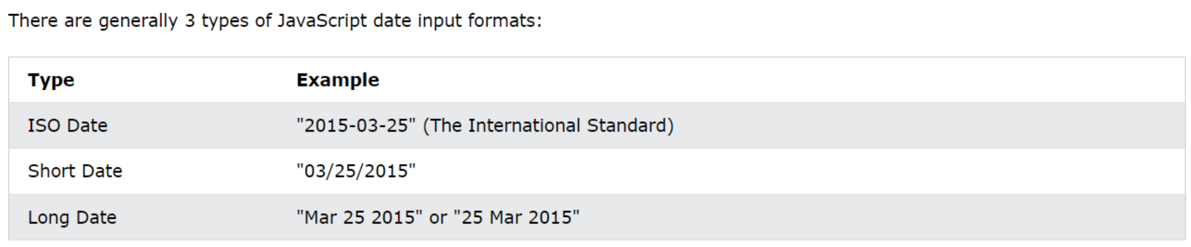
One day (24 hours) are 86 400 000 milliseconds.

Now the time is: **1708014804351** milliseconds past January 01, 1970

new Date(milliseconds)

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

Date Formats



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"

ISO 8601 is the international standard for the representation of dates and times.

The ISO 8601 syntax (YYYY-MM-DD) is also the preferred JavaScript date format:

Complete Date:

```
const d = new Date("2015-03-25");
```

```
Resutlt:    Wed Mar 25 2015 05:30:00 GMT+0530 (India Standard Time)
```

ISO dates can be written without specifying the day (YYYY-MM):

```
const d = new Date("2015-03");
```

ISO dates can be written without month and day (YYYY):

```
const d = new Date("2015");
```

ISO dates can be written with added hours, minutes, and seconds (YYYY-MM-DDTHH:MM:SSZ):

```
const d = new Date("2015-03-25T12:00:00Z");
```

Date and time are separated with a capital T.

UTC time is defined with a capital letter Z.

If you want to modify the time relative to UTC, remove the Z and add +HH:MM or -HH:MM

Short Dates

Short dates are written with an "MM/DD/YYYY" syntax like this:

```
const d = new Date("03/25/2015");
```

Long Dates

Long dates are most often written with an "MMM DD YYYY" syntax like this:

```
const d = new Date("Mar 25 2015");
```

Date Input – Parsing Date()

If you have a valid date string, you can use the Date.parse() method to convert it to milliseconds.

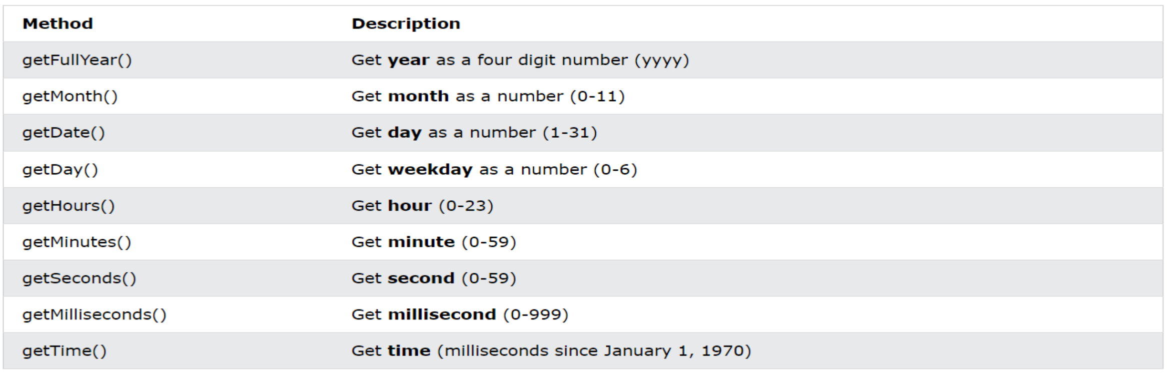
Date.parse() returns the number of milliseconds between the date and January 1, 1970:

```
let msec = Date.parse("March 21, 2012");
```

You can then use the number of milliseconds to **convert it to a date** object:

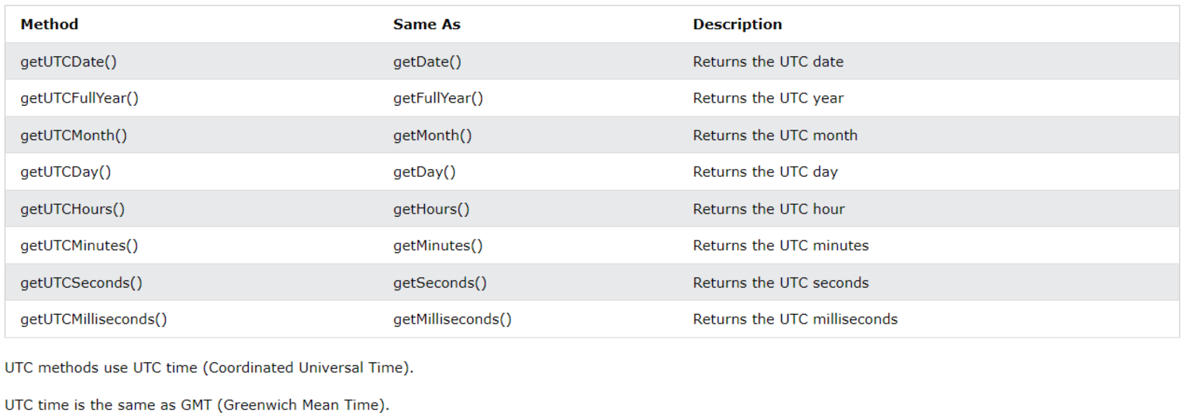
```
let msec = Date.parse("March 21, 2012");
const d = new Date(msec);
```

Date Get 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)

UTC date Methods

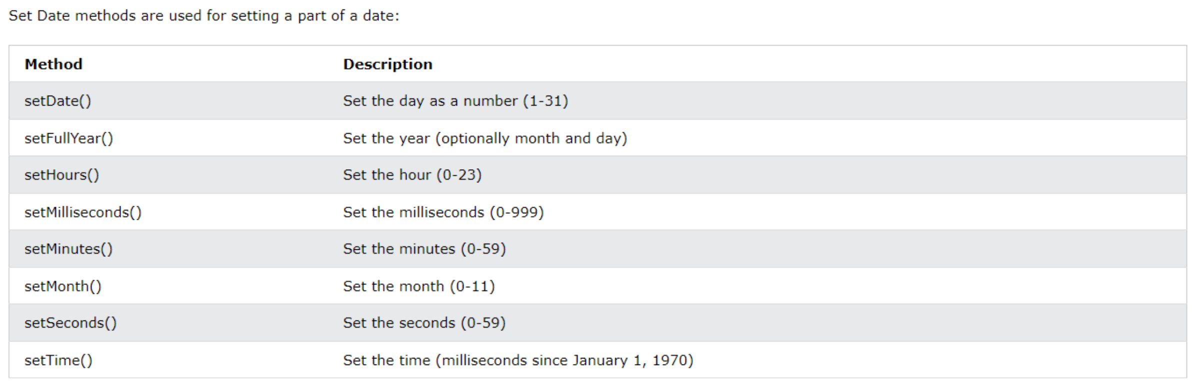


Method	Same As	Description
getUTCDate()	getDate()	Returns the UTC date
getUTCFullYear()	getFullYear()	Returns the UTC year
getUTCMonth()	getMonth()	Returns the UTC month
getUTCDay()	getDay()	Returns the UTC day
getUTCHours()	getHours()	Returns the UTC hour
getUTCMinutes()	getMinutes()	Returns the UTC minutes
getUTCSeconds()	getSeconds()	Returns the UTC seconds
getUTCMilliseconds()	getMilliseconds()	Returns the UTC milliseconds

UTC methods use UTC time (Coordinated Universal Time).

UTC time is the same as GMT (Greenwich Mean Time).

Date Set Methods



Set Date methods are used for setting a part of a date:

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)

Js Math

Def

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

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 property is : Math.*property*.

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

```
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
```

Method	**Description**
abs(x)	Returns the absolute value of x
acos(x)	Returns the arccosine of x, in radians
acosh(x)	Returns the hyperbolic arccosine of x
asin(x)	Returns the arcsine of x, in radians
asinh(x)	Returns the hyperbolic arcsine of x
atan(x)	Returns the arctangent of x as a numeric value between -PI/2 and
atan2(y, x)	Returns the arctangent of the quotient of its arguments

atanh(x)	Returns the hyperbolic arctangent of x
cbrt(x)	Returns the cubic root of x
ceil(x)	Returns x, rounded upwards to the nearest integer
cos(x)	Returns the cosine of x (x is in radians)
cosh(x)	Returns the hyperbolic cosine of x
exp(x)	Returns the value of E^x
floor(x)	Returns x, rounded downwards to the nearest integer
log(x)	Returns the natural logarithm (base E) of x
max(x, y, z, ..., n)	Returns the number with the highest value
min(x, y, z, ..., n)	Returns the number with the lowest value
pow(x, y)	Returns the value of x to the power of y
random()	Returns a random number between 0 and 1
round(x)	Rounds x to the nearest integer
sign(x)	Returns if x is negative, null or positive (-1, 0, 1)
sin(x)	Returns the sine of x (x is in radians)

sinh(x)	Returns the hyperbolic sine of x
sqrt(x)	Returns the square root of x
tan(x)	Returns the tangent of an angle
tanh(x)	Returns the hyperbolic tangent of a number
trunc(x)	Returns the integer part of a number (x)

Js Random

Def

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

```
Math.random()          0.2783255885763676
```

Math.random() used with Math.floor() can be used to return random integers.

```
Math.floor(Math.random() * 10);          5
```

As you can see from the examples above, it might be a good idea to create a proper random function to use for all random integer purposes.

This JavaScript function always returns a random number between min (included) and max (excluded):

```
function getRndInteger(min, max) {
  return Math.floor(Math.random() * (max - min) ) + min;
}
```

This JavaScript function always returns a random number between min and max (both included

```
function getRndInteger(min, max) {
  return Math.floor(Math.random() * (max - min + 1) ) + min;
}
```

Js Boolean

Def

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

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

Boolean(10 > 9)

Or even easier:

(10 > 9)
10 > 9

Booleans as Objects

Normally JavaScript booleans are primitive values created from literals:

let x = false;   // typeof x returns boolean

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

let y = new Boolean(false);   // typeof y returns object

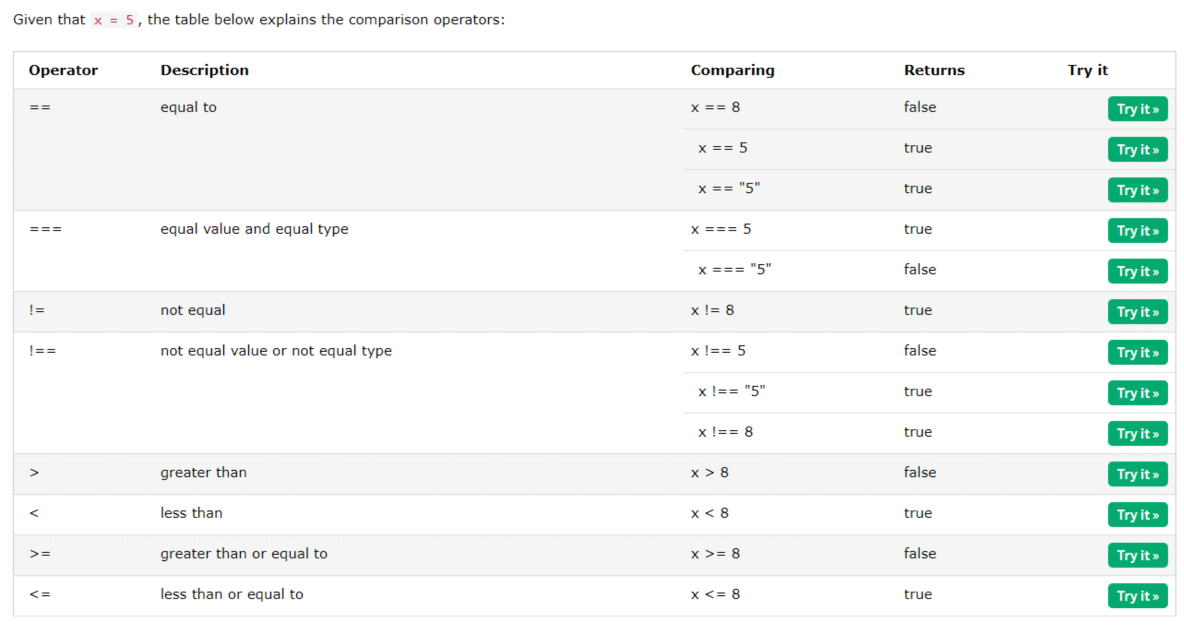
Js Comparison

Def

Comparison and Logical operators are used to test for true or false.

Conditional Operators

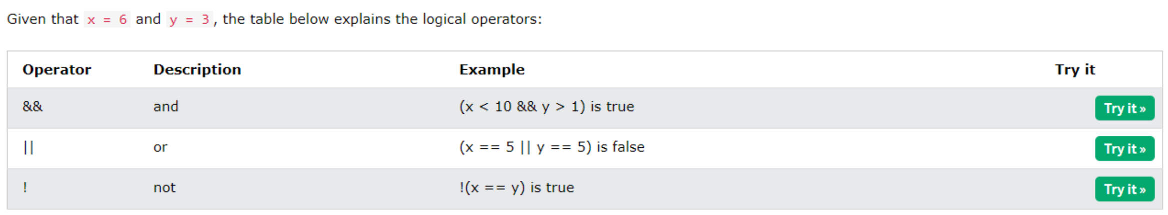
Comparison operators are used to determine equality or difference between variables or values.



Given that `x = 5`, the table below explains the comparison operators:

Operator	Description	Comparing	Returns	Try it
==	equal to	x == 8	false	Try it »
		x == 5	true	Try it »
		x == "5"	true	Try it »
===	equal value and equal type	x === 5	true	Try it »
		x === "5"	false	Try it »
!=	not equal	x != 8	true	Try it »
!==	not equal value or not equal type	x !== 5	false	Try it »
		x !== "5"	true	Try it »
		x !== 8	true	Try it »
>	greater than	x > 8	false	Try it »
<	less than	x < 8	true	Try it »
>=	greater than or equal to	x >= 8	false	Try it »
<=	less than or equal to	x <= 8	true	Try it »

Logical Operators



Given that `x = 6` and `y = 3`, the table below explains the logical operators:

Operator	Description	Example	Try it
&&	and	(x < 10 && y > 1) is true	Try it »
||	or	(x == 5 || y == 5) is false	Try it »
!	not	!(x == y) is true	Try it »

Conditional (Ternary) Operator

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

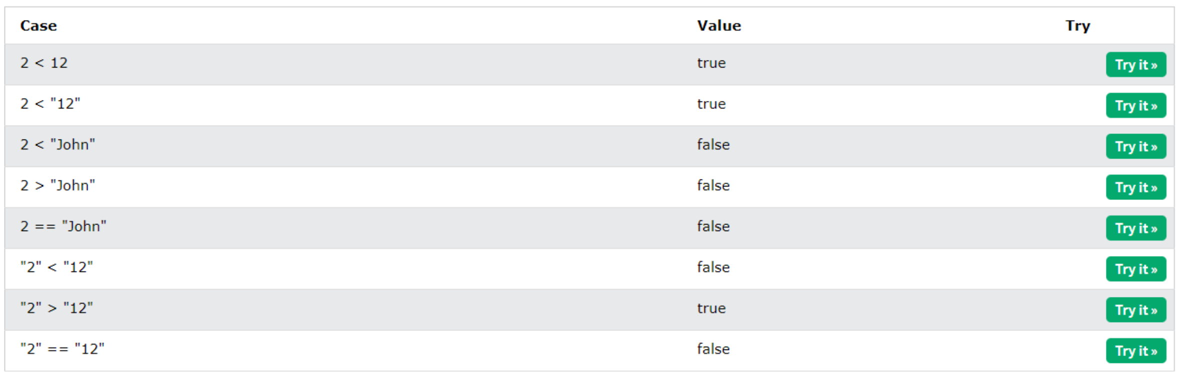
Syntax

variablename = (condition) ? value1:value2

Example

let voteable = (age < 18) ? "Too young":"Old enough";

Comparing String and Number



Case	Value	Try
2 < 12	true	Try it »
2 < "12"	true	Try it »
2 < "John"	false	Try it »
2 > "John"	false	Try it »
2 == "John"	false	Try it »
"2" < "12"	false	Try it »
"2" > "12"	true	Try it »
"2" == "12"	false	Try it »

The Nullish Coalescing Operator (??)

The ?? operator returns the first argument if it is not **nullish** (null or undefined).

Otherwise it returns the second argument.

let name = null;
let text = "missing";
let result = name ?? text;

The Optional Chaining Operator (?.)

The ?. operator returns undefined if an object is undefined or null (instead of throwing an error).

```
// Create an object:
const car = {type:"Fiat", model:"500", color:"white"};
// Ask for car name:
document.getElementById("demo").innerHTML = car?.name;
```

If Else

In JavaScript we 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

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

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

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

Syntax

```
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
}
```

Switch Case

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

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.

Switch cases use strict comparison (===).

The values must be of the same type to match.

break keyword

When JavaScript reaches a break keyword, it breaks out of the switch block.

default keyword

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

The default case does not have to be the last case in a switch block:

Common Code Blocks

Sometimes you will want different switch cases to use the same code.

In this example case 4 and 5 share the same code block, and 0 and 6 share another code block:

```
switch (new Date().getDay()) {
  case 4:
  case 5:
             text = "Soon it is Weekend";
              break;
  case 0:
  case 6:
               text = "It is Weekend";
               break;
  default:
               text = "Looking forward to the Weekend";
}
```

Js Loops

Def

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

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 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

for

The for statement creates a loop with 3 optional expressions:

```
for (expression 1; expression 2; expression 3) {
  // code block to be executed
}
```

Expression 1 is executed (one time) before the execution of the code block.

Expression 2 defines the condition for executing the code block.

Expression 3 is executed (every time) after the code block has been executed.

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

Expression 1,2

You can initiate many values in expression 1 (separated by comma):

Normally you will use expression 1 to initialize the variable used in the loop (let i = 0).

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

And you can omit expression 1 (like when your values are set before the loop starts)

If expression 2 returns true, the loop will start over again. If it returns false, the loop will end.

Expression 3

Expression 3 can do anything like negative increment (i--), positive increment (i = i + 15), or anything else.

Expression 3 can also be omitted (like when you increment your values inside the loop):

```
for (; i < len; )
```

for in

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

```
for (key in object) {
  // code block to be executed
}
```

Example

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

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

Do not use **for in** over an Array if the index **order** is important.

The index order is implementation-dependent, and array values may not be accessed in the order you expect.

It is better to use a **for** loop, a **for of** loop, or **Array.forEach()** when the order is important.

for of

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:

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.

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

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

while

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
}
```

Example

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

do while

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);
```

Example

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

break and continue

The break statement "jumps out" of a loop.

The continue statement "jumps over" one iteration in the loop.

Break

The break statement can also be used to jump out of a loop

```
for (let i = 0; i < 10; i++) {
  if (i === 3) { break; }
  text += "The number is " + i + "<br>"; }
```

Continue

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

```
if (i === 3) { continue; }
```

Js Set

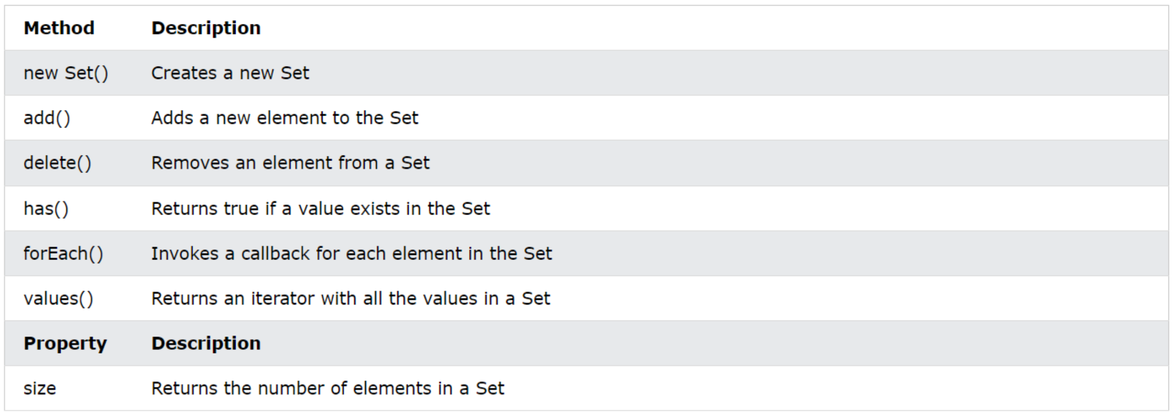
Def

A JavaScript Set is a collection of unique values.

Each value can only occur once in a Set.

You can create a JavaScript Set by:

- Passing an Array to new Set()
- Create a new Set and use add() to add values
- Create a new Set and use add() to add variables



Method	Description
new Set()	Creates a new Set
add()	Adds a new element to the Set
delete()	Removes an element from a Set
has()	Returns true if a value exists in the Set
forEach()	Invokes a callback for each element in the Set
values()	Returns an iterator with all the values in a Set
Property	**Description**
size	Returns the number of elements in a Set

Set Methods

new set()

Pass an Array to the new Set() constructor:

```
// Create a Set
const letters = new Set(["a","b","c"]);
```

Or

Create a Set and add values:

```
// Create a Set
const letters = new Set();
// Add Values to the Set
letters.add("a");
letters.add("b");
letters.add("c");
```

add()

If you add equal elements, only the first will be saved:

```
letters.add("c");
letters.add("c");
```

forEach()

The forEach() method invokes (calls) a function for each Set element:

```
// Create a Set
const letters = new Set(["a","b","c"]);
// List all Elements
let text = "";
letters.forEach (function(value) {
  text += value;
})
```

values()

The values() method returns a new iterator object containing all the values in a Set:

```
letters.values()   // Returns [object Set Iterator]
```

has()

The has() method returns true if a key exists in a Set:

```
fruits.has("apples");
```

delete()

The delete() method removes a Set element:

```
fruits.delete("apples");
```

size

The size property returns the number of elements in a set:

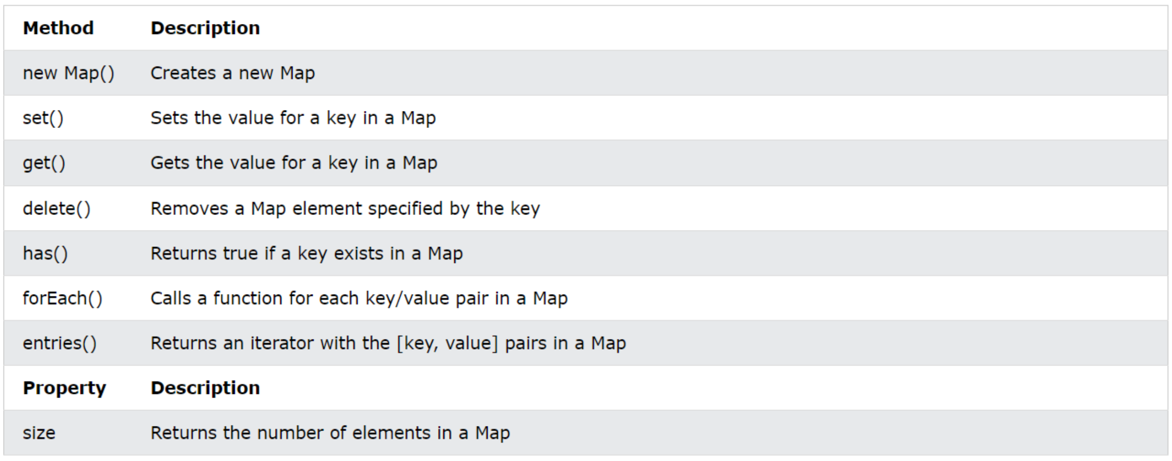
```
fruits.size;
```

Js Maps

Def

A Map holds key-value pairs where the keys can be any datatype.

A Map remembers the original insertion order of the keys.



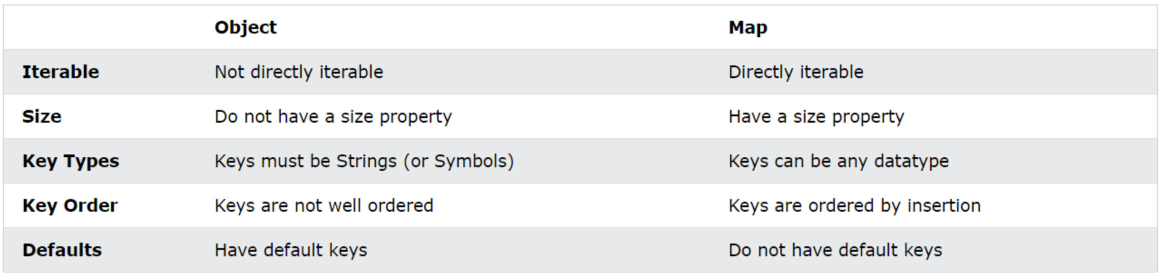
Method	**Description**
new Map()	Creates a new Map
set()	Sets the value for a key in a Map
get()	Gets the value for a key in a Map
delete()	Removes a Map element specified by the key
has()	Returns true if a key exists in a Map
forEach()	Calls a function for each key/value pair in a Map
entries()	Returns an iterator with the [key, value] pairs in a Map
Property	**Description**
size	Returns the number of elements in a Map

You can create a JavaScript Map by:

- Passing an Array to new Map()
- Create a Map and use Map.set()

Object Vs Map

Differences between JavaScript Objects and Maps:



	Object	**Map**
Iterable	Not directly iterable	Directly iterable
Size	Do not have a size property	Have a size property
Key Types	Keys must be Strings (or Symbols)	Keys can be any datatype
Key Order	Keys are not well ordered	Keys are ordered by insertion
Defaults	Have default keys	Do not have default keys

Map Methods

new map()

You can create a Map by passing an Array to the new Map() constructor:

```
// Create a Map
const fruits = new Map([
  ["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);
```

set()

You can add elements to a Map with the set() method:

```
// Create a Map
const fruits = new Map();
// Set Map Values
fruits.set("apples", 500);
fruits.set("bananas", 300);
fruits.set("oranges", 200);
```

get()

The get() method gets the value of a key in a Map:

```
fruits.get("apples");    // Returns 500
```

ForEach()

The forEach() method calls a function for each key/value pair in a Map:

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

entries()

The entries() method returns an iterator object with the [key, values] in a Map:

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

Js TypeOf and Type Conversions

Def

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

TypeOf

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

The TypeOf 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).

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]}
```

Please observe:

- 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** *

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

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.

Undefined

In JavaScript, a variable without a value, has the value undefined. The type is also undefined.

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

Any variable can be emptied, by setting the value to undefined. The type will also be undefined.

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

Null

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

Unfortunately, in JavaScript, the data type of null is an object.

You can consider it a bug in JavaScript that typeof null is an object. It should be null.

You can empty an object by setting it to null:

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

Undefined Vs Null

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

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

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

Instance Of

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);
```

Void

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>
```

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

These are the type conversion in javaScript:

1. Converting Strings to Numbers
2. Converting Numbers to Strings
3. Converting Dates to Numbers
4. Converting Numbers to Dates
5. Converting Booleans to Numbers
6. Converting Numbers to Booleans

Automatic Type Conversion

When JavaScript tries to operate on a "wrong" data type, it will try to convert the value to a "right" type.

The result is not always what you expect:

```
5 + null    // returns 5         because null is converted to 0
"5" + null  // returns "5null"   because null is converted to "null"
"5" + 2     // returns "52"      because 2 is converted to "2"
"5" - 2     // returns 3         because "5" is converted to 5
"5" * "2"   // returns 10        because "5" and "2" are converted to 5 and 2
```

Original Value	Converted to Numbers	Converted to Strings	Converted to Boolean	Try it
false	0	"false"	false	Try it »
true	1	"true"	true	Try it »
0	0	"0"	false	Try it »
1	1	"1"	true	Try it »
"0"	0	"0"	**true**	Try it »
"000"	0	"000"	**true**	Try it »
"1"	1	"1"	true	Try it »
NaN	NaN	"NaN"	false	Try it »

Infinity	Infinity	"Infinity"	true	Try it »
-Infinity	-Infinity	"-Infinity"	true	Try it »
""	**0**	""	**false**	Try it »
"20"	20	"20"	true	Try it »
"twenty"	NaN	"twenty"	true	Try it »
[ ]	**0**	""	true	Try it »
[20]	**20**	"20"	true	Try it »
[10,20]	NaN	"10,20"	true	Try it »
["twenty"]	NaN	"twenty"	true	Try it »
["ten","twenty"]	NaN	"ten,twenty"	true	Try it »
function(){}	NaN	"function(){}"	true	Try it »
{ }	NaN	"[object Object]"	true	Try it »
null	**0**	"null"	false	Try it »
undefined	NaN	"undefined"	false	

Values in quotes indicate string values.

Red values indicate values (some) programmers might not expect.

Js Bitwise Operator

Def

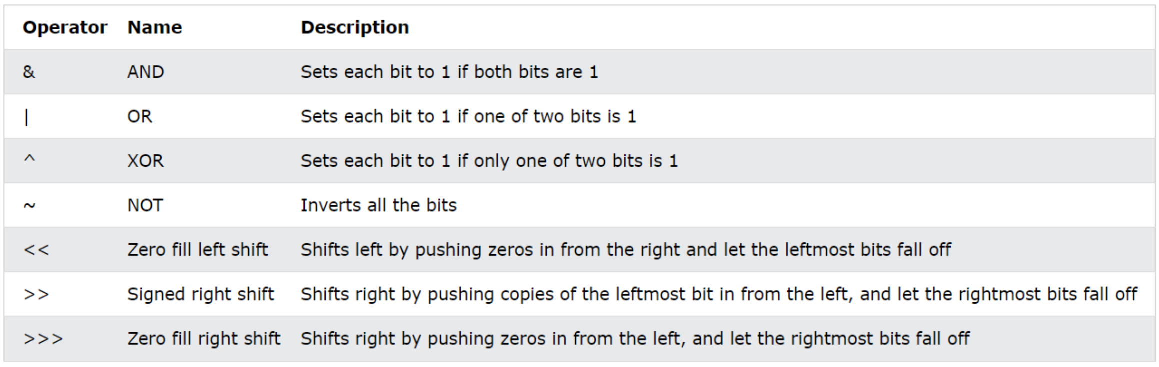
JavaScript stores numbers as 64 bits floating point numbers, but all bitwise operations are performed on 32 bits binary numbers.

Before a bitwise operation is performed, it converts numbers to 32 bits signed integers.

After the bitwise operation is performed, the result is converted back to 64 bits numbers.

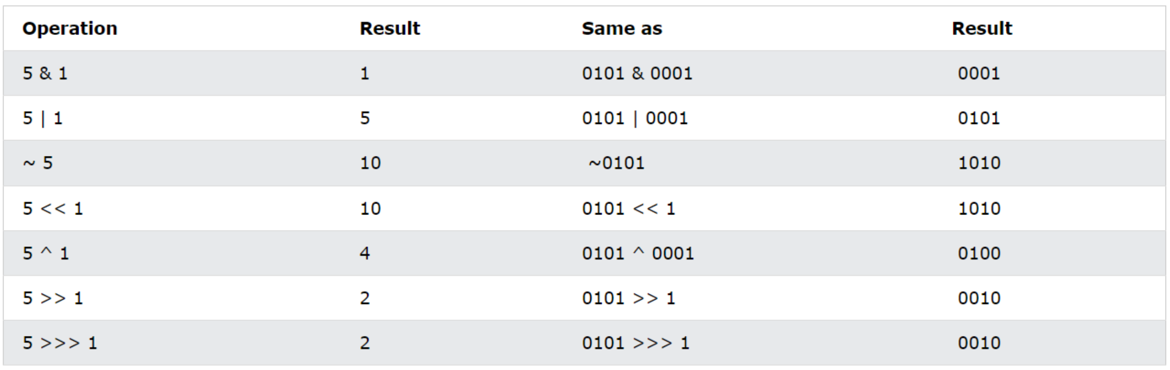
JavaScript binary numbers are stored in two's complement format.

This means that a negative number is the bitwise NOT of the number plus 1:



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

Examples



Operation	Result	Same as	Result
5 & 1	1	0101 & 0001	0001
5 | 1	5	0101 | 0001	0101
~ 5	10	~0101	1010
5 << 1	10	0101 << 1	1010
5 ^ 1	4	0101 ^ 0001	0100
5 >> 1	2	0101 >> 1	0010
5 >>> 1	2	0101 >>> 1	0010

Js RegExp

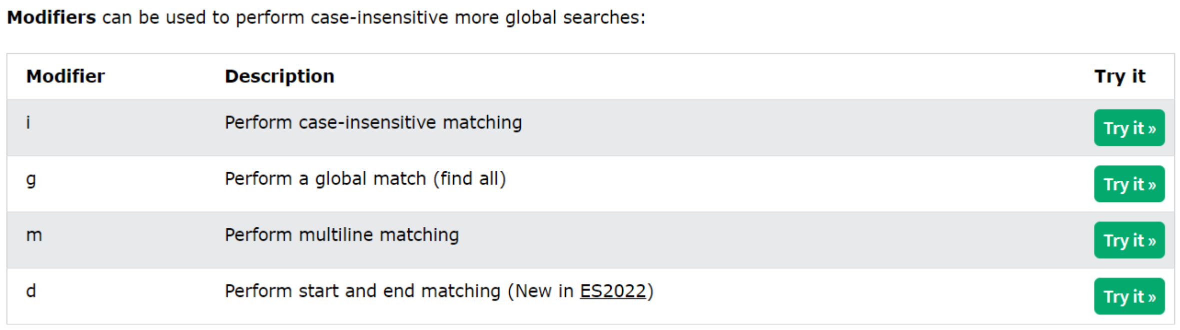
Def

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.

/pattern/modifiers;

Modifiers

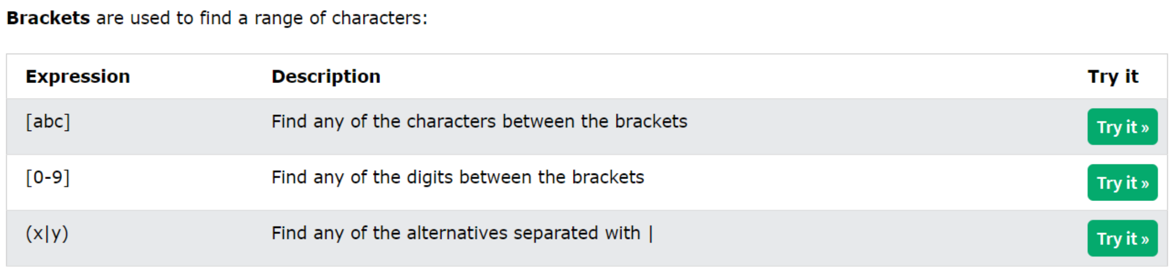


Modifiers can be used to perform case-insensitive more global searches:

Modifier	Description	Try it
i	Perform case-insensitive matching	Try it »
g	Perform a global match (find all)	Try it »
m	Perform multiline matching	Try it »
d	Perform start and end matching (New in ES2022)	Try it »

Patterns

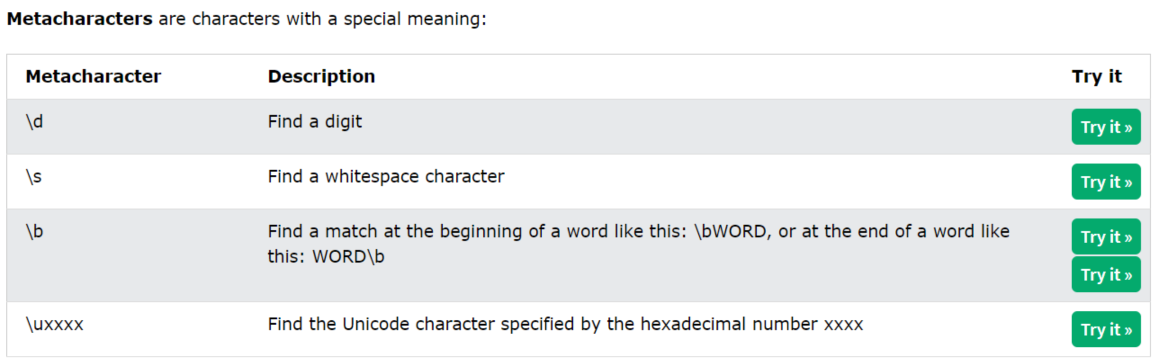
Brackets



Brackets are used to find a range of characters:

Expression	Description	Try it
[abc]	Find any of the characters between the brackets	Try it »
[0-9]	Find any of the digits between the brackets	Try it »
(x|y)	Find any of the alternatives separated with |	Try it »

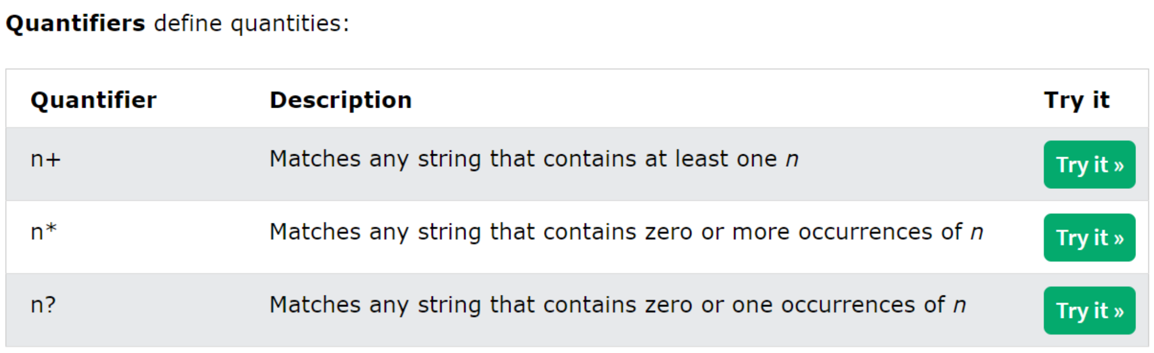
Meta Characters



Metacharacters are characters with a special meaning:

Metacharacter	Description	Try it
\d	Find a digit	Try it »
\s	Find a whitespace character	Try it »
\b	Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b	Try it » Try it »
\uxxxx	Find the Unicode character specified by the hexadecimal number xxxx	Try it »

Quantifiers



Quantifiers define quantities:

Quantifier	Description	Try it
n+	Matches any string that contains at least one *n*	Try it »
n*	Matches any string that contains zero or more occurrences of *n*	Try it »
n?	Matches any string that contains zero or one occurrences of *n*	Try it »

Methods

test()

The test() method is a RegExp expression method.

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

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

exec()

The exec() method is a RegExp expression method.

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

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

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

Js Operator Precedence

Expressions in parentheses are computed **before** the rest of the expression Function are executed **before** the result is used in the rest of the expression			
Val	Operator	Description	Example
18	( )	Expression Grouping	(100 + 50) * 3
17	.	Member Of	person.name
17	[]	Member Of	person["name"]
17	?.	Optional Chaining ES2020	x ?. y
17	()	Function Call	myFunction()
17	new	New with Arguments	new Date("June 5,2022")
16	new	New without Arguments	new Date()
Increment Operators Postfix increments are executed **before** prefix increments			

15	++	Postfix Increment	i++
15	--	Postfix Decrement	i--
14	++	Prefix Increment	++i
14	--	Prefix Decrement	--i
NOT Operators			
14	!	Logical NOT	!(x==y)
14	~	Bitwise NOT	~x
Unary Operators			
14	+	Unary Plus	+x
14	-	Unary Minus	-x
14	typeof	Data Type	typeof x
14	void	Evaluate Void	void(0)
14	delete	Property Delete	delete myCar.color
Arithmetic Operators Exponentiations are executed **before** multiplications Multiplications and divisions are executed **before** additions and subtractions			
13	**	Exponentiation ES2016	10 ** 2
12	*	Multiplication	10 * 5
12	/	Division	10 / 5
12	%	Division Remainder	10 % 5
11	+	Addition	10 + 5

11	-	Subtraction	10 - 5
11	+	Concatenation	"John" + "Doe"
Shift Operators			
10	<<	Shift Left	x << 2
10	>>	Shift Right (signed)	x >> 2
10	>>>	Shift Right (unsigned)	x >>> 2
Relational Operators			
9	in	Property in Object	"PI" in Math
9	instanceof	Instance of Object	x instanceof Array
Comparison Operators			
9	<	Less than	x < y
9	<=	Less than or equal	x <= y
9	>	Greater than	x > y
9	>=	Greater than or equal	x >= Array
8	==	Equal	x == y
8	===	Strict equal	x === y
8	!=	Unequal	x != y
8	!==	Strict unequal	x !== y
Bitwise Operators			

7	&	Bitwise AND	x & y
6	^	Bitwise XOR	x ^ y
5	|	Bitwise OR	x | y
Logical Operators			
4	&&	Logical AND	x && y
3	||	Logical OR	x || y
3	??	Nullish Coalescing ES2020	x ?? y
Conditional (ternary) Operator			
2	? :	Condition	? "yes" : "no"
Assignment Operators Assignments are executed **after** other operations			
2	=	Simple Assignment	x = y
2	:	Colon Assignment	x: 5
2	+=	Addition Assignment	x += y
2	-=	Subtraction Assignment	x -= y
2	*=	Multiplication Assignment	x *= y
2	**=	Exponentiation Assignment	x **= y
2	/=	Division Assignment	x /= y
2	%=	Remainder Assignment	x %= y
2	<<=	Left Shift Assignment	x <<= y

2	>>=	Right Shift Assignment	x >>= y
2	>>>=	Unsigned Right Shift	x >>>= y
2	&=	Bitwise AND Assignment	x &= y
2	|=	Bitwise OR Assignment	x |= y
2	^=	Bitwise XOR Assignment	x ^= y
2	&&=	Logical AND Assignment	x &&= y
2	||=	Logical OR Assignment	x ||= y
2	=>	Arrow	x => y
2	yield	Pause / Resume	yield x
2	yield*	Delegate	yield* x
2	...	Spread	... x
1	,	Comma	x , y

Js Errors

Def

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.

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
}
```

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.

Example

Input Validation

```
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;
  }
```

Html Validation

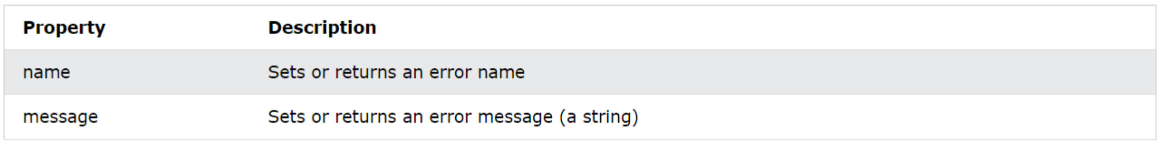
```
<input id="demo" type="number" min="5" max="10" step="1">
```

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.

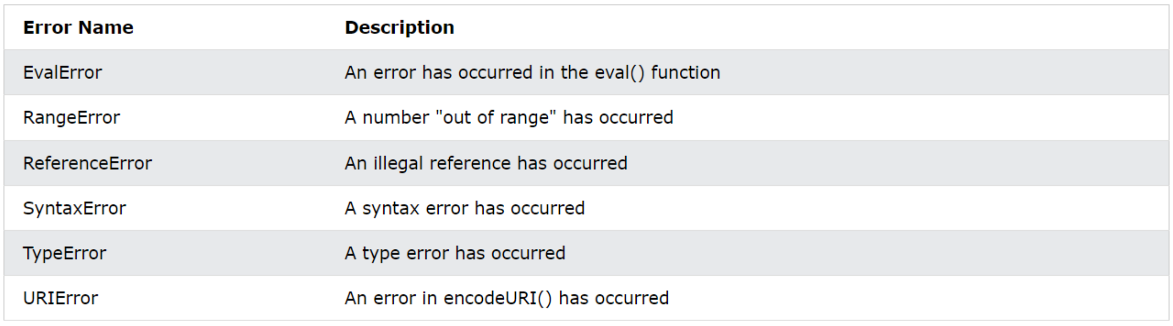
Properties



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

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

Js Hoisting

Var Hoisting

Variables defined with var are hoisted to the top and can be initialized at any time.

Meaning: You can use the variable before it is declared:

```
carName = "Volvo";

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

var carName;
```

Let Hoisting

Variables defined with let are also hoisted to the top of the block, but not initialized.

Meaning: Using a let variable before it is declared will result in a ReferenceError.

```
try {

  carName = "Saab";

  let carName = "Volvo";

}

catch(err) {
  document.getElementById("demo").innerHTML = err;
}
```

With **let**, you cannot use a variable before it is declared.

ReferenceError: Cannot access 'carName' before initialization

Const Hoisting

Using a const variable before it is declared, is a syntax error, so the code will simply not run.

This code will not run.

```
carName = "Volvo";
const carName;
```

With **const**, you cannot use a variable before it is declared.

SyntaxError: Cannot access 'carName' before initialization

Initializations are Not Hoisted

JavaScript only hoists declarations, not initializations.

Example 1 does **not** give the same result as **Example 2**:

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
```

This is because only the declaration (var y), not the initialization (=7) is hoisted to the top.

Because of hoisting, y has been declared before it is used, but because initializations are not hoisted, the value of y is undefined.

Hoisting is (to many developers) an unknown or overlooked behavior of JavaScript.

If a developer doesn't understand hoisting, programs may contain bugs (errors).

To avoid bugs, always declare all variables at the beginning of every scope.

Since this is how JavaScript interprets the code, it is always a good rule.

Js Strict Mode

Def

The "use strict" directive was new in ECMAScript version 5.

It is not a statement, but a literal expression, ign ored by earlier versions of JavaScript.

The purpose of "use strict" is to indicate that the code should be executed in "strict mode".

The "use strict" directive is only recognized at the **beginning** of a script or a function.

You can use strict mode in all your programs. It helps you to write cleaner code, like preventing you from using undeclared variables.

"use strict" is just a string, so IE 9 will not throw an error even if it does not understand it.

Declaring Strict Mode

Strict mode is declared by adding "use strict"; to the beginning of a script or a function.

Declared at the beginning of a script, it has global scope (all code in the script will execute in strict mode):

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

Why Strict Mode?

Strict mode makes it easier to write "secure" JavaScript.

Strict mode changes previously accepted "bad syntax" into real errors.

As an example, in normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, making it impossible to accidentally create a global variable.

In normal JavaScript, a developer will not receive any error feedback assigning values to non-writable properties.

In strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.

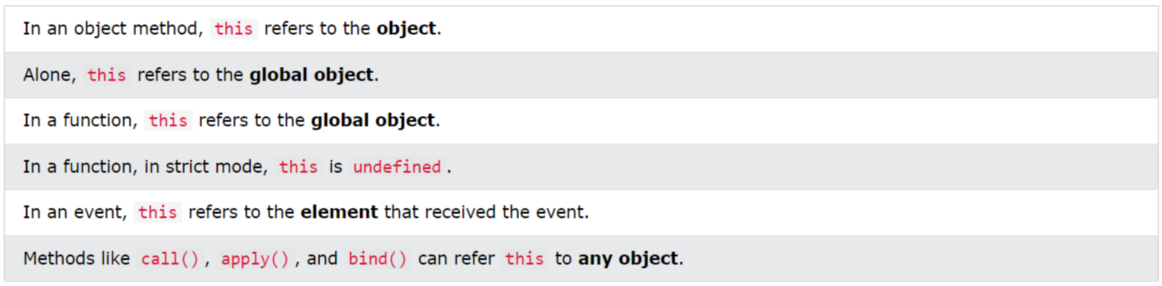
This Keyword

Def

In JavaScript, the this keyword refers to an **object**.

Which object depends on how this is being invoked (used or called).

The this keyword refers to different objects depending on how it is used:



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**.

Example

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

Explicit Function Binding

The call() and apply() methods are predefined JavaScript methods.

They can both be used to call an object method with another object as argument.

The example below calls person1.fullName with person2 as an argument, **this** refers to person2, even if fullName is a method of person1:

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

Function Borrowing

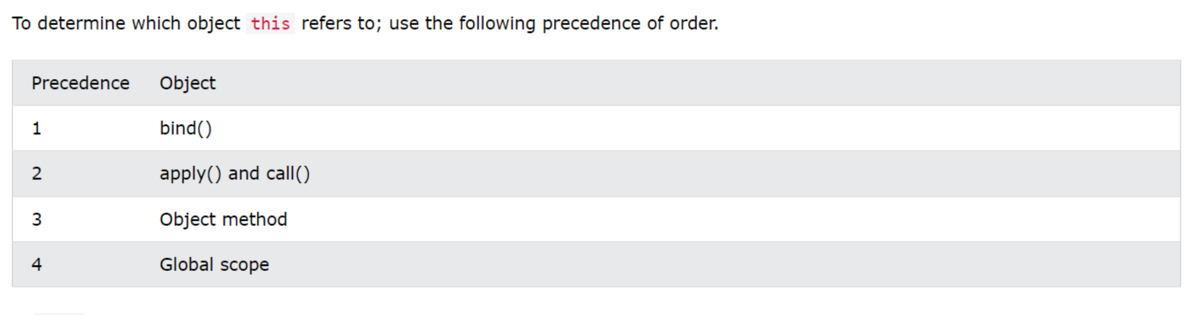
With the bind() method, an object can borrow a method from another object.

This example 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);
```

This Precedence



To determine which object `this` refers to; use the following precedence of order.

Precedence	Object
1	bind()
2	apply() and call()
3	Object method
4	Global scope

Arrow Function

Def

Arrow functions were introduced in ES6.

Arrow functions allow us to write shorter function syntax:

let myFunction = (a, b) => a * b;

Before Arrow:

```
hello = function() {
  return "Hello World!";
}
```

With Arrow Function:

```
hello = () => {
  return "Hello World!";
}
```

Other ways

Arrow Functions Return Value by Default:

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

Arrow Function With Parameters:

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

Arrow Function Without Parentheses:

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

This Keyword and Arrow Function

The handling of this is also different in arrow functions compared to regular functions.

In short, with arrow functions there are no binding of this.

In regular functions the this keyword represented the object that called the function, which could be the window, the document, a button or whatever.

With arrow functions the this keyword *always* represents the object that defined the arrow function.

Js Classes

Def

ECMAScript 2015, also known as ES6, introduced JavaScript Classes.

JavaScript Classes are templates for JavaScript Objects.

Syntax

Use the keyword class to create a class.

Always add a method named constructor():

```
class ClassName {
  constructor() { ... }
}
```

Constructor Method

The constructor method is a special method:

- It has to have the exact name "constructor"
- It is executed automatically when a new object is created
- It is used to initialize object properties

If you do not define a constructor method, JavaScript will add an empty constructor method.

Class Method

Class methods are created with the same syntax as object methods.

Use the keyword class to create a class.

Always add a constructor() method.

Then add any number of methods.

```
class ClassName {
  constructor() { ... }
  method_1() { ... }
}
```

Example

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;         }
  age(x) {
    return x - this.year;
  }}
const date = new Date();
let year = date.getFullYear();
const myCar = new Car("Ford", 2014);
document.getElementById("demo").innerHTML=
"My car is " + myCar.age(year) + " years old.";
```

Js Modules

Def

JavaScript modules allow you to break up your code into separate files.

This 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>
```

Exports

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

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

Named Exports

Let us create a file named person.js, and fill it with the things we want to export.

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

In-line individually:

```
person.js

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

All at once at the bottom:

```
person.js

const name = "Jesse";
const age = 40;
export {name, age};
```

Default Exports

Let us create another file, named message.js, and use it for demonstrating default export.

You can only have one default export in a file.

```
message.js

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

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";

Js Objects – Advanced

Def

In JavaScript, 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.

Objects are Variables

JavaScript variables can contain single values:

```
let person = "John Doe";
```

JavaScript variables can also contain many values.

Objects are variables too. But objects can contain many values.

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

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

Object Properties

The named values, in JavaScript objects, are called **properties**.

Property	Value
firstName	John
lastName	Doe
age	50
eyeColor	blue

Objects written as name value pairs are similar to:

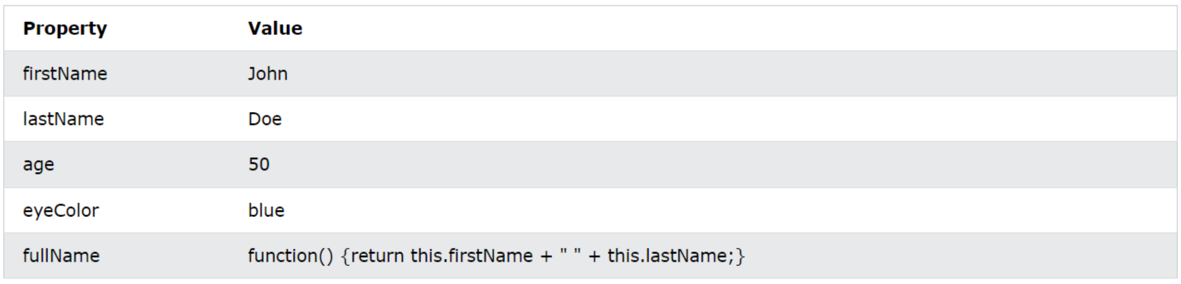
- Associative arrays in PHP
- Dictionaries in Python
- Hash tables in C
- Hash maps in Java
- Hashes in Ruby and Perl

Object Methods

Methods are **actions** that can be performed on objects.

Object properties can be both primitive values, other objects, and functions.

An **object method** is an object property containing a **function definition**.



Property	Value
firstName	John
lastName	Doe
age	50
eyeColor	blue
fullName	function() {return this.firstName + " " + this.lastName;}

Creating a JavaScript Object

With JavaScript, you can define and create your own objects.

There are different ways to create new objects:

- Create a single object, using an object literal.
- Create a single object, with the keyword new.
- Define an object constructor, and then create objects of the constructed type.
- Create an object using Object.create().

Using the JavaScript Keyword new

The following example create a new JavaScript object using new Object(), and then adds 4 properties:

```
const person = new Object();
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

JavaScript Objects are Mutable

Objects are mutable: They are addressed by reference, not by value.

If person is an object, the following statement will not create a copy of person:

const x = person; // Will not create a copy of person.

The object x is **not a copy** of person. It **is** person. Both x and person are the same object.

Any changes to x will also change person, because x and person are the same object.

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

const x = person;
x.age = 10;      // Will change both x.age and person.age
```

Object Properties

Properties are the values associated with a JavaScript object.

A JavaScript object is a collection of unordered properties.

Properties can usually be changed, added, and deleted, but some are read only.

Accessing JavaScript Properties

The syntax for accessing the property of an object is:

```
objectName.property     // person.age

                   or

objectName["property"]  // person["age"]

                   or

objectName[expression]  // x = "age"; person[x]
```

The expression must evaluate to a property name.

The JavaScript for...in statement loops through the properties of an object.

The block of code inside of the for...in loop will be executed once for each property.

Looping through the properties of an object:

```
const person = {
  fname:" John",
  lname:" Doe",
  age: 25
};
for (let x in person) {
  txt += person[x];
}
```

Adding New Properties to Object

You can add new properties to an existing object by simply giving it a value.

Assume that the person object already exists - you can then give it new properties:

```
person.nationality = "English";
```

Deleting Properties

The delete keyword deletes a property from an object:

```
delete person.age;
```

Or

```
delete person["age"];
```

The delete keyword deletes both the value of the property and the property itself.

After deletion, the property cannot be used before it is added back again.

The delete operator is designed to be used on object properties. It has no effect on variables or functions.

The delete operator should not be used on predefined JavaScript object properties. It can crash your application.

Property Attributes

All properties have a name. In addition, they also have a value.

The value is one of the property's attributes.

Other attributes are: enumerable, configurable, and writable.

These attributes define how the property can be accessed (is it readable? is it writable?)

In JavaScript, all attributes can be read, but only the value attribute can be changed (and only if the property is writable).

(ECMAScript 5 has methods for both getting and setting all property attributes)

Prototype Properties

JavaScript objects inherit the properties of their prototype.

The delete keyword does not delete inherited properties, but if you delete a prototype property, it will affect all objects inherited from the prototype.

Object Methods

JavaScript methods are actions that can be performed on objects.

A JavaScript **method** is a property containing a **function definition**.

Property	Value
firstName	John
lastName	Doe
age	50
eyeColor	blue
fullName	function() {return this.firstName + " " + this.lastName;}

Accessing Object Methods

You access an object method with the following syntax:

```
objectName.methodName()
```

You will typically describe fullName() as a method of the person object, and fullName as a property.

The fullName property will execute (as a function) when it is invoked with ().

This example accesses the fullName() **method** of a person object:

```
name = person.fullName();
```

If you access the fullName **property**, without (), it will return the **function definition**:

```
name = person.fullName;
```

Adding a Method to an Object

Adding a new method to an object is easy:

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

Object Display

Displaying a JavaScript object will output **[object Object]**.

Some common solutions to display JavaScript objects are:

- Displaying the Object Properties by name
- Displaying the Object Properties in a Loop
- Displaying the Object using Object.values()
- Displaying the Object using JSON.stringify()

The properties of an object can be displayed as a string:

```
document.getElementById("demo").innerHTML =
person.name + "," + person.age + "," + person.city;
```

The properties of an object can be collected in a loop:

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

document.getElementById("demo").innerHTML = txt;
```

Any JavaScript object can be converted to an array using Object.values():

```
const myArray = Object.values(person);
document.getElementById("demo").innerHTML = myArray;
```

Any JavaScript object can be stringified (converted to a string) with the JavaScript function JSON.stringify():

```
let myString = JSON.stringify(person);
document.getElementById("demo").innerHTML = myString;
```

Object Accessors (Getters and Setters)

ECMAScript 5 (ES5 2009) introduced Getter and Setters.

Getters and setters allow you to define Object Accessors (Computed Properties).

Getter (The get Keyword)

This example uses a lang property to get the value of the language property.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  language: "en",
  get lang() {
    return this.language;
  }};
// Display data from the object using a getter:
document.getElementById("demo").innerHTML = person.lang;
```

Setter (The set Keyword)

This example uses a lang property to set the value of the language property.

```
const person = {
  firstName: "John",
  language: "",
  set lang(lang) {
    this.language = lang;
  }};
// Set an object property using a setter:
person.lang = "en";


// Display data from the object:
document.getElementById("demo").innerHTML = person.language;
```

Data Quality by Object Accessors

JavaScript can secure better data quality when using getters and setters.

Using the lang property, returns the value of the language property in upper case:

```
get lang() {
  return this.language.toUpperCase();
 }
```

Why Using Getters and Setters?

- 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 Constructors

The examples from the previous chapters are limited. They only create single objects.

Sometimes we need a "**blueprint**" for creating many objects of the same "type".

The way to create an "object type", is to use an **object constructor function**.

The function Person() is an object constructor function.

Objects of the same type are created by calling the constructor function with the new keyword:

```
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}

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

Adding a Property to a Constructor

You cannot add a new property to an object constructor the same way you add a new property to an existing object:

```
Person.nationality = "English";
```

To add a new property to a constructor, you must add it to the constructor function:

```
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyecolor;
  this.nationality = "English";
}
```

This way object properties can have default values.

Adding a Method to a Constructor

Your constructor function can also define methods:

```
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyecolor;
  this.name = function() {
    return this.firstName + " " + this.lastName;
  };}
```

You cannot add a new method to an object constructor the same way you add a new method to an existing object.

Adding methods to an object constructor must be done inside the constructor function:

```
function Person(firstName, lastName, age, eyeColor) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.age = age;
  this.eyeColor = eyeColor;
  this.changeName = function (name) {
    this.lastName = name;
  };}
```

The changeName() function assigns the value of name to the person's lastName property.

Now You Can Try:

```
myMother.changeName("Doe");
```

Built-in JavaScript Constructors

JavaScript has built-in constructors for native objects:

```
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
```

Object Prototypes

All JavaScript objects inherit properties and methods from a prototype:

- Date objects inherit from Date.prototype
- Array objects inherit from Array.prototype
- Person objects inherit from Person.prototype

The Object.prototype is on the top of the prototype inheritance chain:

Date objects, Array objects, and Person objects inherit from Object.prototype.

Sometimes you want to add new properties (or methods) to all existing objects.

Sometimes you want to add new properties (or methods) to an object constructor.

Using the prototype Property

The JavaScript prototype property allows you to add new properties to object constructors:

The JavaScript prototype property also allows you to add new methods to objects constructors:

```
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyecolor;
}
Person.prototype.name = function() {
  return this.firstName + " " + this.lastName;
};
```

Only modify your **own** prototypes. Never modify the prototypes of standard JavaScript objects.

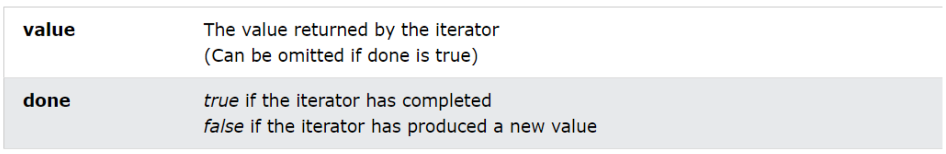
JavaScript Iterators

The **iterator protocol** defines how to produce a **sequence of values** from an object.

An object becomes an **iterator** when it implements a next() method.

The next() method must return an object with two properties:

- value (the next value)
- done (true or false)



value	The value returned by the iterator (Can be omitted if done is true)
done	*true* if the iterator has completed *false* if the iterator has produced a new value

Home – Made Iterators

This iterable returns never ending: 10,20,30,40,.... Everytime next() is called:

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

// Create Iterable
const n = myNumbers();
n.next(); // Returns 10
n.next(); // Returns 20
n.next(); // Returns 30
```

The problem with a home made iterable:

It does not support the JavaScript for..of statement.

A JavaScript iterable is an object that has a **Symbol.iterator**.

The Symbol.iterator is a function that returns a next() function.

An iterable can be iterated over with the code: for (const x of iterable) { }

```
// 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};
 }  }; }
```

Now you can use for..of

```
for (const num of myNumbers) {
  // Any Code Here
}
```

The Symbol.iterator method is called automatically by for..of.

But we can also do it "manually":

```
let iterator = myNumbers[Symbol.iterator]();
while (true) {
  const result = iterator.next();
  if (result.done) break;
  // Any Code Here
}
```

Object - Sets

A JavaScript Set is a collection of unique values.

Each value can only occur once in a Set.

A Set can hold any value of any data type.

Method	Description
new Set()	Creates a new Set
add()	Adds a new element to the Set
delete()	Removes an element from a Set
has()	Returns true if a value exists
clear()	Removes all elements from a Set
forEach()	Invokes a callback for each element
values()	Returns an Iterator with all the values in a Set
keys()	Same as values()
entries()	Returns an Iterator with the [value,value] pairs from a Set

Property	Description
size	Returns the number elements in a Set

How to Create a Set

You can create a JavaScript Set by:

- Passing an Array to new Set()
- Create a new Set and use add() to add values
- Create a new Set and use add() to add variables

Set() Method

Pass an Array to the new Set() constructor:

```
// Create a Set

const letters = new Set(["a","b","c"]);
```

Create a Set and add literal values:

```
// Create a Set
```

```
const letters = new Set();

// Add Values to the Set

letters.add("a");

letters.add("b");

letters.add("c");
```

If you add equal elements, only the first will be saved:

forEach() Method

The forEach() method invokes a function for each Set element:

```
// Create a Set
const letters = new Set(["a","b","c"]);
// List all entries
let text = "";
letters.forEach (function(value) {
  text += value;
})
```

values() Method

The values() method returns an Iterator object containing all the values in a Set:

Now you can use the Iterator object to access the elements:

```
// Create an Iterator
const myIterator = letters.values();
// List all Values
let text = "";
for (const entry of myIterator) {
  text += entry;
}
```

keys() Method

A Set has no keys.

keys() returns the same as values().

This makes Sets compatible with Maps.

```
letters.keys()   // Returns [object Set Iterator]
```

entries() Method

A Set has no keys.

entries() returns [value,value] pairs instead of [key,value] pairs.

```
const myIterator = letters.entries();
```

Sets are Objects

For a Set, typeof returns object:

```
typeof letters;      // Returns object
```

For a Set, instanceof Set returns true:

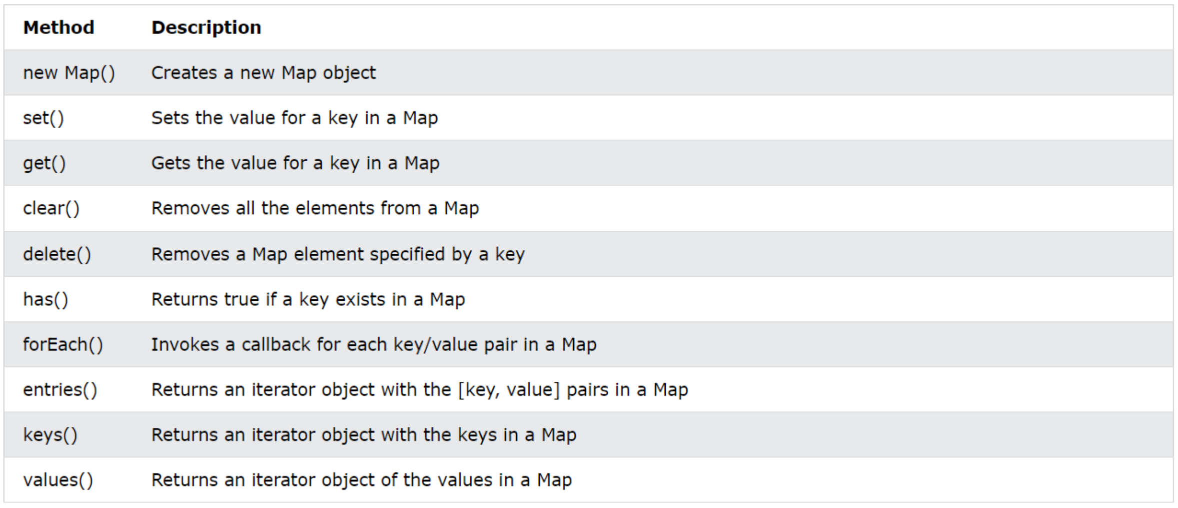
```
letters instanceof Set;  // Returns true
```

Objects – Maps

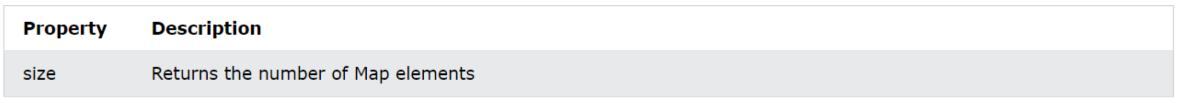
A Map holds key-value pairs where the keys can be any datatype.

A Map remembers the original insertion order of the keys.

A Map has a property that represents the size of the map.



Method	Description
new Map()	Creates a new Map object
set()	Sets the value for a key in a Map
get()	Gets the value for a key in a Map
clear()	Removes all the elements from a Map
delete()	Removes a Map element specified by a key
has()	Returns true if a key exists in a Map
forEach()	Invokes a callback for each key/value pair in a Map
entries()	Returns an iterator object with the [key, value] pairs in a Map
keys()	Returns an iterator object with the keys in a Map
values()	Returns an iterator object of the values in a Map



Property	Description
size	Returns the number of Map elements

How to Create a Map

You can create a JavaScript Map by:

- Passing an Array to new Map()
- Create a Map and use Map.set()

Map()

You can create a Map by passing an Array to the new Map() constructor:

```
// Create a Map
const fruits = new Map([
  ["apples", 500],
  ["bananas", 300],
  ["oranges", 200]   ]);
```

Map.set()

You can add elements to a Map with the set() method:

```
// Create a Map
const fruits = new Map();
// Set Map Values
fruits.set("apples", 500);
```

Map.get()

The get() method gets the value of a key in a Map:

```
fruits.get("apples");    // Returns 500
```

Map.size

The size property returns the number of elements in a Map:

```
fruits.size;
```

Map.delete()

The delete() method removes a Map element:

```
fruits.delete("apples");
```

Map.clear()

The clear() method removes all the elements from a Map:

```
fruits.clear();
```

Map.has()

The has() method returns true if a key exists in a Map:

```
fruits.has("apples");
```

Maps are Objects

typeof returns object:

```
// Returns object:
typeof fruits;
```

instanceof Map returns true:

```
// Returns true:
fruits instanceof Map;
```

Object Vs Maps

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

Object Reference

Managing Objects

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

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

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

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

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

// Accessing the prototype
Object.getPrototypeOf(object)

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

Protecting Objects

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

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

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

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

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

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

Changing a Property Value

Object.defineProperty(object, property, {value : *value*})

This example changes a property value:

```
const person = {
  firstName: "John",
  lastName : "Doe",
  language : "EN"
};
// Change a property
Object.defineProperty(person, "language", {value : "NO"})
```

Changing Meta Data

ES5 allows the following property meta data to be changed:

```
writable : true      // Property value can be changed
enumerable : true    // Property can be enumerated
configurable : true  // Property can be reconfigured

writable : false     // Property value can not be changed
enumerable : false   // Property can be not enumerated
configurable : false // Property can be not reconfigured
```

ES5 allows getters and setters to be changed:

```
// Defining a getter
get: function() { return language }
// Defining a setter
set: function(value) { language = value }
```

This example makes language read-only:

```
Object.defineProperty(person, "language", {writable:false});
```

This example makes language not enumerable:

```
Object.defineProperty(person, "language", {enumerable:false});
```

Listing All Properties

This example list all properties of an object:

```
const person = {
  firstName: "John",
  lastName : "Doe",
  language : "EN"
};
Object.defineProperty(person, "language", {enumerable:false});
Object.getOwnPropertyNames(person);  // Returns an array of properties
```

This example list only the enumerable properties of an object:

```
Object.keys(person);  // Returns an array of enumerable properties
```

Adding a Property

This example adds a new property to an object:

```
// Create an object:
const person = {
  firstName: "John",
  lastName : "Doe",
  language : "EN"
};
// Add a property
Object.defineProperty(person, "year", {value:"2008"});
```

Adding Getters and Setters

The Object.defineProperty() method can also be used to add Getters and Setters:

```
//Create an object
const person = {firstName:"John", lastName:"Doe"};

// Define a getter
Object.defineProperty(person, "fullName", {
  get: function () {return this.firstName + " " + this.lastName;}
});
```

Js Functions

Def

JavaScript functions are **defined** with the function keyword.

You can use a function **declaration** or a function **expression**.

Function Declarations

Earlier in this tutorial, you learned that functions are **declared** with the following syntax:

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

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are invoked (called upon).

```
function myFunction(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);
```

The function above is actually an **anonymous function** (a function without a name).

Functions stored in variables do not need function names. They are always invoked (called) using the variable name.

The Function() Constructor

As you have seen in the previous examples, JavaScript functions are defined with the function keyword.

Functions can also be defined with a built-in JavaScript function constructor called Function().

```
const myFunction = new Function("a", "b", "return a * b");
let x = myFunction(4, 3);
```

You actually don't have to use the function constructor. The example above is same as writing:

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

Function Hoisting

Hoisting is default behavior of moving **declarations** to the top of the current scope.

Hoisting applies to variable declarations and to function declarations.

Because of this, JavaScript functions can be called before they are declared:

```
myFunction(5);
function myFunction(y) {
  return y * y;
}
```

Functions defined using an expression are not hoisted.

Self-Invoking Functions

Function expressions can be made "self-invoking".

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:

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

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

Functions are Objects

The typeof operator in JavaScript returns "function" for functions.

But, JavaScript functions can best be described as objects.

JavaScript functions have both **properties** and **methods**.

The arguments.length property returns the number of arguments received when the function was invoked:

```
function myFunction(a, b) {
  return arguments.length;
}
```

A function defined as the property of an object, is called a method to the object.
A function designed to create new objects, is called an object constructor.

Arrow Functions

Arrow functions allows a short syntax for writing function expressions.

You don't need the function keyword, the return keyword, and the **curly brackets**.

```
// ES5
var x = function(x, y) {
  return x * y;
}
// ES6
const x = (x, y) => x * y;
```

Arrow functions do not have their own this. They are not well suited for defining **object methods**.

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.

You can only omit the return keyword and the curly brackets if the function is a single statement. Because of this, it might be a good habit to always keep them:

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

Function Parameters

A JavaScript function does not perform any checking on parameter values (arguments).

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

Function **parameters** are the **names** listed in the function definition.

Function **arguments** are the real **values** passed to (and received by) the function.

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.

Default Parameters

If a function is called with **missing arguments** (less than declared), the missing values are set to undefined.

Sometimes this is acceptable, but sometimes it is better to assign a default value to the parameter:

```
function myFunction(x, y) {
  if (y === undefined) {
    y = 2;
  }}
```

Default Parameter Values

ES6 allows function parameters to have default values.

If y is not passed or undefined, then y = 10.

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

Function Rest Parameter

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

```
function sum(...args) {
  let sum = 0;
  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).

This way you can simply use a function to find (for instance) the highest value in a list of numbers:

```
x = findMax(1, 123, 500, 115, 44, 88);
function findMax() {
  let max = -Infinity;
  for (let i = 0; i < arguments.length; i++) {
    if (arguments[i] > max) {
      max = arguments[i];
    } }
  return max; }
```

Arguments are Passed by Value

The parameters, in a function call, are the function's arguments.

JavaScript arguments are passed by **value**: The function only gets to know the values, not the argument's locations.

If a function changes an argument's value, it does not change the parameter's original value.

Changes to arguments are not visible (reflected) outside the function.

Objects are Passed by Reference

In JavaScript, object references are values.

Because of this, objects will behave like they are passed by **reference:**

If a function changes an object property, it changes the original value.

Changes to object properties are visible (reflected) outside the function.

Function Invocation

There are three types of Function Invocations:

1. Invoking Functions as Functions
2. Invoking Functions as Methods
3. Invoking Functions as Constructors

Invoking Functions as Functions

The function above does not belong to any object. But in JavaScript there is always a default global object.

In HTML the default global object is the HTML page itself, so the function above "belongs" to the HTML page.

In a browser the page object is the browser window. The function above automatically becomes a window function.

```
function myFunction(a, b) {
  return a * b;     }
myFunction(10, 2);         // Will return 20
```

This is a common way to invoke a JavaScript function, but not a very good practice. Global variables, methods, or functions can easily create name conflicts and bugs in the global object.

myFunction() and window.myFunction() is the same function:

```
function myFunction(a, b) {
  return a * b;
}
window.myFunction(10, 2);    // Will also return 20
```

Invoking a Function as a Method

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

Invoking a Function with a Function Constructor

If a function invocation is preceded with the new keyword, it is a constructor invocation.

It looks like you create a new function, but since JavaScript functions are objects you actually create a new object:

```
// This is a function constructor:
function myFunction(arg1, arg2) {
  this.firstName = arg1;
  this.lastName  = arg2;
}
// This creates a new object
const myObj = new myFunction("John", "Doe");
// This will return "John"
myObj.firstName;
```

A constructor invocation creates a new object. The new object inherits the properties and methods from its constructor.

Functions Call, Bind, and Apply

Call() Method

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.

This example calls the **fullName** method of person, using it on **person1**:

The call() method can accept 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");
```

Apply() Method

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

The apply() method is similar to the call() method (previous chapter).

In this example the **fullName** method of **person** is **applied** on **person1**:

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

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

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

call() Method vs apply() Method

The difference is:

The call() method takes arguments **separately**.

The apply() method takes arguments as an **array**.

The apply() method is very handy if you want to use an array instead of an argument list.

Bind() Method

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 the bind() method has to be used to prevent losing **this**.

When a function is used as a callback, **this** is lost.

This example will try to display the person name after 3 seconds, but it will display **undefined** instead:

```
setTimeout(person.display, 3000);
```

The bind() method solves this problem.

In the following example, the bind() method is used to bind person.display to person.

This example will display the person name after 3 seconds:

```
let display = person.display.bind(person);
setTimeout(display, 3000);
```

Function Closures

JavaScript variables can belong to the **local** or **global** scope.

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

Counter Dilemma

Suppose you want to use a variable for counting something, and you want this counter to be available to all functions.

You could use a global variable, and a function to increase the counter:

```
// Initiate counter
let counter = 0;
// Function to increment counter
function add() {
  counter += 1;
}
// Call add() 3 times
add();
add();
add();
// The counter should now be 3
```

There is a problem with the solution above: Any code on the page can change the counter, without calling add().

The counter should be local to the add() function, to prevent other code from changing it:

```
// Function to increment counter
function add() {
  let counter = 0;
  counter += 1;
}
```

It did not work because we display the global counter instead of the local counter.

We can remove the global counter and access the local counter by letting function return it:

```
// Function to increment counter
function add() {
  let counter = 0;
  counter += 1;
  return counter;
}
```

It did not work because we reset the local counter every time we call the function.

A JavaScript inner function can solve this.

JavaScript Nested Functions

All functions have access to the global scope.

In fact, in JavaScript, all functions have access to the scope "above" them.

JavaScript supports nested functions. Nested functions have access to the scope "above" them.

The inner function plus() has access to the counter variable in the parent function:

```
function add() {
  let counter = 0;
  function plus() {counter += 1;}
  plus();
  return counter;    }
```

This could have solved the counter dilemma, if we could reach the plus() function from the outside.

We also need to find a way to execute counter = 0 only once.

We need a closure.

JavaScript Closures

Remember self-invoking functions? What does this function do?

```
const add = (function () {
  let counter = 0;
  return function () {counter += 1; return counter}
})();
add();
add();
add();
// the counter is now 3
```

The variable add is assigned to the return value of a self-invoking function.

The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.

This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope.

This is called a JavaScript **closure.** It makes it possible for a function to have "**private**" variables.

The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

A closure is a function having access to the parent scope, even after the parent function has closed.

Js Classes

Def

ECMAScript 2015, also known as ES6, introduced JavaScript Classes.

JavaScript Classes are templates for JavaScript Objects.

Use the keyword class to create a class.

Always add a method named constructor():

Syntax

```
class ClassName {
  constructor() { ... }
}
```

Example

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

The example above creates a class named "Car".

The class has two initial properties: "name" and "year".

A JavaScript class is **not** an object.

It is a **template** for JavaScript objects.

When you have a class, you can use the class to create objects:

```
const myCar1 = new Car("Ford", 2014);
const myCar2 = new Car("Audi", 2019);
```

The Constructor Method

The constructor method is a special method:

- It has to have the exact name "constructor"
- It is executed automatically when a new object is created
- It is used to initialize object properties

If you do not define a constructor method, JavaScript will add an empty constructor method.

Class Methods

Class methods are created with the same syntax as object methods.

Use the keyword class to create a class.

Always add a constructor() method.

Then add any number of methods.

```
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.";
```

You can send parameters to Class methods:

Class Inheritance

To create a class inheritance, use the extends keyword.

A class created with a class inheritance inherits all the methods from another class:

Create a class named "Model" which will inherit the methods from the "Car" class:

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return 'I have a ' + this.carname;
  } }
class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a ' + this.model;
  } }
let myCar = new Model("Ford", "Mustang");
document.getElementById("demo").innerHTML = myCar.show();
```

The super() method refers to the parent class.

By calling the super() method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.

Getters and Setters

Classes also allows you to use getters and setters.

It can be smart to use getters and setters for your properties, especially if you want to do something special with the value before returning them, or before you set them.

To add getters and setters in the class, use the get and set keywords.

Create a getter and a setter for the "carname" property:

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  get cnam() {
    return this.carname;
  }
  set cnam(x) {
    this.carname = x;
  } }
const myCar = new Car("Ford");
document.getElementById("demo").innerHTML = myCar.cnam;
```

Note: even if the getter is a method, you do not use parentheses when you want to get the property value.

The name of the getter/setter method cannot be the same as the name of the property, in this case carname.

Many programmers use an underscore character _ before the property name to separate the getter/setter from the actual property:

You can use the underscore character to separate the getter/setter from the actual property:

```
class Car {
  constructor(brand) {
    this._carname = brand;
  }
  get carname() {
    return this._carname;
  }
  set carname(x) {
    this._carname = x;
  } }
const myCar = new Car("Ford");
document.getElementById("demo").innerHTML = myCar.carname;
```

To use a *setter*, use the same syntax as when you set a property value, without parentheses:

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

Hoisting

Unlike functions, and other JavaScript declarations, class declarations are not hoisted.

That means that you must declare a class before you can use it:

Static Methods

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.
```

If you want to use the myCar object inside the static method, you can send it as a parameter:

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello(x) {
    return "Hello " + x.name;
  }
}
const myCar = new Car("Ford");
document.getElementById("demo").innerHTML = Car.hello(myCar);
```