**Code structure:**

**Statements:**

Statements are syntax constructs and commands that perform actions.

For example:

alert('Hello'); alert('World');

**Semicolons:**

A semicolon may be omitted in most cases when a line break exists.

For ex:

alert('Hello')

alert('World')

**Comments:**

Single Line comment:

// This comment occupies a line of its own

alert('Hello');

alert('World'); // This comment follows the statement

multiline Comment:

/\* An example with two messages.

This is a multiline comment.

\*/

**Use strict Mode:**

"use strict"; Defines that JavaScript code should be executed in "strict mode".

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

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

With strict mode, you can not, for example, use undeclared variables.

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.

**Variable:**

A variable is a “named storage” for data. We can use variables to store goodies, visitors, and other data.

Create variable using: Let

For example

Ex. let x;//declare variable

X=10//store value or assign value

Multiple variable in single line:

Ex: let user = “Megha” , subject=”js”;

Create variable using: var=> The var keyword is almost the same as let. It also declares a variable

Ex : Var message = “hello world”

Difference between let and var We can re-declare variable while use var but not doing same while use let:

We cannot re-declare variable in let

Ex let x=10;

Let x=20;//show error

**[Constants](https://javascript.info/variables" \l "constants):** Variables declared using const are called “constants”. They cannot be reassigned ex: const pi = 3.14;

# **Data types:**

A value in JavaScript is always of a certain type. For example, a string or a number.Programming languages that allow such things, such as JavaScript, are called “dynamically typed”, meaning that there exist data types, but variables are not bound to any of them.

Ex: let x = 10; x = “hello”

**1.Number=>**

The number type represents both integer and floating point numbers.

There are many operations for numbers, e.g. multiplication \*, division /, addition +, subtraction -, and so on.Besides regular numbers, there are so-called “special numeric values” which also belong to this data type: Infinity, -Infinity and NaN.

Infinity represents the mathematical Infinity ∞. It is a special value that’s greater than any number.

Ex: alert(1/0) ;

NaN represents a computational error. It is a result of an incorrect or an undefined mathematical operation, for instance:

alert( "not a number" / 2 );

**2.BigInt=>**

In JavaScript, the “number” type cannot safely represent integer values larger than (253-1) (that’s 9007199254740991), or less than -(253-1) for negatives.

Example const data= 787798787987n;

**3.String=>**

A string in JavaScript must be surrounded by quotes.

Ex: let name= “megha”;

**3.Boolen=>**

The boolean type has only two values: true and false.

This type is commonly used to store yes/no values: true means “yes, correct”, and false means “no, incorrect”.

Ex: let isGreter = 4>1;

Alert(isGreter);

**4.The null value=>**

The special null value does not belong to any of the types described above.

Let marks= null;

**5.The undefined value=>**

Ex:let marks;

Alert(marks);

**6.Objects and Symbols=>**

The object type is special.

All other types are called “primitive” because their values can contain only a single thing (be it a string or a number or whatever). In contrast, objects are used to store collections of data and more complex entities. Being that important, objects deserve a special treatment. We’ll deal with them later in the chapter Objects, after we learn more about primitives. The symbol type is used to create unique identifiers for objects. We have to mention it here for the sake of completeness, but also postpone the details till we know objects.

**6.The type of operator=>**

The typeof operator returns the type of the operand. It’s useful when we want to process values of different types differently or just want to do a quick check.

typeof undefined // "undefined"

typeof 0 // "number"

typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"

typeof Math // "object" (1)

typeof null // "object" (2)

typeof alert // "function" (3)

**Interaction: alert, prompt, confirm**

**1.alert**

shows a message.

Ex:alert(“hello world”);

**2.Prompt:**

shows a message asking the user to input text. It returns the text or, if Cancel button or Esc is clicked, null.

The function prompt accepts two arguments:

result = prompt(title, [default]);

It shows a modal window with a text message, an input field for the visitor, and the buttons OK/Cancel.

**title**

The text to show the visitor.

**default**

An optional second parameter, the initial value for the input field.

**2.Prompt:**

shows a message and waits for the user to press “OK” or “Cancel”. It returns true for OK and false for Cancel/Esc.

result = confirm(question);

The function confirm shows a modal window with a question and two buttons: OK and Cancel.

The result is true if OK is pressed and false otherwise.

# **Type Conversions**

**1. String Conversion:**

String conversion happens when we need the string form of a value.

For example, alert(value) does it to show the value.

We can also call the String(value) function to convert a value to a string:

Example:

let value = true;

alert(typeof value); // boolean

value = String(value); // now value is a string "true"

alert(typeof value); /

**2. Numeric Conversion:**

Numeric conversion in mathematical functions and expressions happens automatically.

For example, when division / is applied to non-numbers:

let str = "123";

alert(typeof str); // string

let num = Number(str); // becomes a number 123

alert(typeof num); // number

Example:

alert( Number(" 123 ") ); // 123

alert( Number("123z") ); // NaN (error reading a number at "z")

alert( Number(true) ); // 1

alert( Number(false) ); // 0

**2. Boolean Conversion:** Boolean conversion is the simplest one.

It happens in logical operations (later we’ll meet condition tests and other similar things) but can also be performed explicitly with a call to Boolean(value).

The conversion rule:

Values that are intuitively “empty”, like 0, an empty string, null, undefined, and NaN, become false.

Other values become true.

Example:

alert( Boolean(1) ); // true

alert( Boolean(0) ); // false

alert( Boolean("hello") ); // true

alert( Boolean("") ); // false

alert( Boolean("0") ); // true

alert( Boolean(" ") ); // spaces, also true (any non-empty string is true)

# **Basic operators, maths:**

**1.Math:**

The following math operations are supported:

Addition +,

Subtraction -,

Multiplication \*,

Division /,

Remainder %,

Exponentiation \*\*.

**Example:**

**Reminder:**

alert( 5 % 2 ); // 1, the remainder of 5 divided by 2

**Exponentiation :**

alert( 2 \*\* 2 ); // 2² = 4

**String concatenation with binary +**

let s = "my" + "string";

alert(s); // mystring

**Numeric conversion, unary +**

let x = 1;

alert( +x ); // 1

let y = -2;

alert( +y ); // -2

// Converts non-numbers

alert( +true ); // 1

alert( +"" ); // 0

**2. Assignment**

an assignment = is also an operator. It is listed in the precedence table with the very low priority of 2.

Example:

let x = 2 \* 2 + 1;

alert( x ); // 5

**Chaining assignments**

Example:

let a, b, c;

a = b = c = 2 + 2;

alert( a ); // 4

alert( b ); // 4

alert( c ); // 4

**Modify-in-place**

**Eample:**

let n = 2;

n += 5; // now n = 7 (same as n = n + 5)

n \*= 2; // now n = 14 (same as n = n \* 2)

alert( n ); // 14

**3.Increment/decrement**

**Increment:**

let counter = 2;

counter++; // works the same as counter = counter + 1, but is shorter

alert( counter ); // 3

**Decrement:**

let counter = 2;

counter--; // works the same as counter = counter - 1, but is shorter

alert( counter ); // 1

**4. Bitwise operators:**

The list of operators:

AND ( & )

OR ( | )

XOR ( ^ )

NOT ( ~ )

LEFT SHIFT ( << )

RIGHT SHIFT ( >> )

ZERO-FILL RIGHT SHIFT ( >>> )

**5. Comma**

The comma operator , is one of the rarest and most unusual operators. Sometimes, it’s used to write shorter code,

Example:

let a = (1 + 2, 3 + 4);

alert( a );

# **Conditional branching: if, '?'**

**The “if” statement**

The if(...) statement evaluates a condition in parentheses and, if the result is true, executes a block of code.

Subtopic AND,OR,NOT

**Nullish coalescing operator '??'**

Example

let user;

alert(user ?? "Anonymous"); // Anonymous (user is undefined)

**The "switch" statement**

switch(x) {

case 'value1': // if (x === 'value1')

...

[break]

case 'value2': // if (x === 'value2')

...

[break]

default:

...

[break]

}

**Loops: while and for:**

**1.while loop:**

while (condition) {

// code

// so-called "loop body"

}

**2.do while loop:**

do {

// loop body

} while (condition);

**3.for loop:**

for (begin; condition; step) {

// ... loop body ...

}

# **Functions**

Functions are the main “building blocks” of the program. They allow the code to be called many times without repetition.

Function declaration

Function parameter

Local variable

Global variable

Function default value

Function as returning value

**Function expressions**

function sayHi() {

alert( "Hello" );

}

There is another syntax for creating a function that is called a Function Expression.

let sayHi = function() {

alert( "Hello" );

};

## [Callback functions](https://javascript.info/function-expressions" \l "callback-functions)

## Let’s look at more examples of passing functions as values and using function expressions.We’ll write a function ask(question, yes, no) with three parameters:

## Question

## Text of the question

## yes

## Function to run if the answer is “Yes”

## no

## Function to run if the answer is “No”

## function askQuestion(que,ans1,ans2){

## if(confirm(que))

## ans1();

## else

## ans2();

## }

## function showOk(){

## alert("data deleted");

## }

## function showCancle(){

## alert("data not deleted");

## }

## askQuestion("You want to delete data????",showOk,showCancle);

## Function Expression vs Function Declaration

Function Declaration: a function, declared as a separate statement

// Function Declaration

function sum(a, b) {

return a + b;

}

Function Expression: a function, created inside an expression or inside another syntax construct. Here, the function is created on the right side of the “assignment expression” =:

// Function Expression

let sum = function(a, b) {

return a + b;

};

# **Arrow functions, the basics**

let func = function(arg1, arg2, ..., argN) {

return expression;

};

**Single line arrow function**

Ex :

Let sum = (a,b) => a+b;

alert(sum(12,30);

**Multiline arrow function**

let sum = (a, b) => {

let result = a + b;

return result;

};

alert( sum(1, 2) ); // 3

**Objects**

objects are used to store keyed collections of various data and more complex entities. In JavaScript, objects penetrate almost every aspect of the language. So we must understand them first before going in-depth anywhere else.

An object can be created with figure brackets {…} with an optional list of properties. A property is a “key: value” pair, where key is a string (also called a “property name”), and value can be anything.

Example:

let user = new Object(); // "object constructor" syntax

let user = {}; // "object literal" syntax

let user = {

name: "John",

age: 30

};

// get property values of the object:

alert( user.name ); // John

alert( user.age ); // 30

# **Object references and copying**

let user = {

name: "John"

};

let admin = user; // copy the reference

**‘This’ in method**

let Product={

pid:100,

pname:"laptop",

//method

showProduct(){

alert("pid="+ this.pid + " productname="+ this.pname);

}

};

Product.showProduct();

**“this” is not bound**

let Product={

pid:100,

pname:"laptop",

//method

showProduct(){

alert("pid="+ this.pid + " productname="+ this.pname);

}

};

Product.showProduct();

//This is not bound

let User={

name:"Megha",

}

let Admin={

name:"Malay"

}

function printName(){

alert(this.name);

}

User.f=printName;

Admin.f=printName;

Admin.f();//Print Malay

User.f();//Print Megha

**Constructor, operator "new"**

The regular {...} syntax allows us to create one object. But often we need to create many similar objects, like multiple users or menu items and so on.

That can be done using constructor functions and the "new" operator.

**Constructor function**

Constructor functions technically are regular functions. There are two conventions though:

They are named with capital letter first.

They should be executed only with "new" operator.

Example:

function User(name) {

this.name = name;

this.isAdmin = false;

}

let user = new User("Jack");

alert(user.name); // Jack

alert(user.isAdmin); // false

**Data types**

JavaScript allows us to work with primitives (strings, numbers, etc.) as if they were objects. They also provide methods to call as such.

the key distinctions between primitives and objects.

**A primitive**

Is a value of a primitive type.

There are 7 primitive types: string, number, bigint, boolean, symbol, null and undefined.

**An object**

Is capable of storing multiple values as properties.

Can be created with {}, for instance: {name: "John", age: 30}. There are other kinds of objects in JavaScript: functions, for example, are objects.

**A primitive as an object**

let str = "Hello";

alert( str.toUpperCase() ); // HELLO

The string str is a primitive. So in the moment of accessing its property, a special object is created that knows the value of the string, and has useful methods, like toUpperCase().

# **Numbers**

Regular numbers in JavaScript are stored in 64-bit format IEEE-754, also known as “double precision floating point numbers”. These are numbers that we’re using most of the time, and we’ll talk about them in this chapter.

BigInt numbers represent integers of arbitrary length. They are sometimes needed because a regular integer number can’t safely exceed (253-1) or be less than -(253-1), as we mentioned earlier in the chapter Data types. As bigints are used in few special areas, we devote them a special chapter BigInt.

Rounding

One of the most used operations when working with numbers is rounding.

There are several built-in functions for rounding:

Math.floor

Rounds down: 3.1 becomes 3, and -1.1 becomes -2.

Math.ceil

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

Math.round

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

Math.trunc (not supported by Internet Explorer)

Removes anything after the decimal point without rounding: 3.1 becomes 3, -1.1 becomes -1.

**Strings**

In JavaScript, the textual data is stored as strings. There is no separate type for a single character.

Example: let single = 'single-quoted';

let double = "double-quoted";

let backticks = `backticks`;

Print Multiline string:

Example:

let guestList = `Guests:

\* John

\* Pete

\* Mary `;

alert(guestList); // a list of guests, multiple lines

**Note:**

let guestList = "Guests: // Error: Unexpected token ILLEGAL

\* John";

**Special characters**

Example:

let guestList = "Guests:\n \* John\n \* Pete\n \* Mary";

alert(guestList);

| **Character** | **Description** |
| --- | --- |
|  |  |
| \n | New line |
| \r | In Windows text files a combination of two characters \r\n represents a new break, while on non-Windows OS it’s just \n. That’s for historical reasons, most Windows software also understands \n. |
| \', \", \` | Quotes |
| \\ | Backslash |
| \t | Tab |
| \b, \f, \v | Backspace, Form Feed, Vertical Tab – mentioned for completeness, coming from old times, not used nowadays (you can forget them right now). |

**Accessing characters**

To get a character at position pos, use square brackets [pos] or call the method str.at(pos). The first character starts from the zero position:

let str = `Hello`;

// the first character

alert( str[0] ); // H

alert( str.at(0) ); // H

// the last character

alert( str[str.length - 1] ); // o

alert( str.at(-1) );

**String function:**

alert( 'Interface'.toUpperCase() ); // INTERFACE

alert( 'Interface'.toLowerCase() ); // interface

let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, because 'Widget' is found at the beginning

alert( str.indexOf('widget') ); // -1, not found, the search is case-sensitive

alert( str.indexOf("id") ); // 1, "id" is found at the position 1 (..idget with id)

The more modern method str.includes(substr, pos) returns true/false depending on whether str contains substr within.

alert( "Widget with id".includes("Widget") ); // true

alert( "Hello".includes("Bye") ); // false

**Array**

Declaration

let arr = new Array();

let arr = [];

let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[0] ); // Apple

alert( fruits[1] ); // Orange

alert( fruits[2] ); // Plum

**array method:**

Methods pop/push, shift/unshift

A queue is one of the most common uses of an array. In computer science, this means an ordered collection of elements which supports two operations:

push appends an element to the end.

shift get an element from the beginning, advancing the queue, so that the 2nd element becomes the 1st.

Push: Append the element to the end of the array:

Pop: Extracts the last element of the array and returns it:

Shift: Extracts the first element of the array and returns it:

Unshift: Add the element to the beginning of the array:

**array slpice**

// delete Fruits[2];//remove value from index

Fruits.splice(1,1);//remove index and value

console.log(Fruits);

**array slice**

let arr = ["t", "e", "s", "t"];

alert( arr.slice(1, 3) );

alert( arr.slice(-2) );

**array concat**

let first=["hello","world"];

let sec=["php","nodejs"];

console.log(first.concat(sec));

**sort**

var numArray = [140000, 104,

numArray.sort(function(a

return a + b;//desc

//return a-b//asc

});

console.log(numArray);

# **Iterables**

Iterable objects are objects that can be iterated over with for..of.

Technically, iterables must implement the Symbol.iterator method.

Example:

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

# **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 |

# **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 |

# **Date and time**

To create a new Date object call new Date() with one of the following arguments:

**new Date()**

**Access date components**

There are methods to access the year, month and so on from the Date object:

getFullYear()

Get the year (4 digits)

getMonth()

Get the month, from 0 to 11.

getDate()

Get the day of month, from 1 to 31, the name of the method does look a little bit strange.

getHours(), getMinutes(), getSeconds(), getMilliseconds()

Get the corresponding time components.

Not getYear(), but getFullYea

# **JSON methods, toJSON**

The JSON (JavaScript Object Notation) is a general format to represent values and objects. It is described as in RFC 4627 standard. Initially it was made for JavaScript, but many other languages have libraries to handle it as well. So it’s easy to use JSON for data exchange when the client uses JavaScript and the server is written on Ruby/PHP/Java/Whateve

JSON.stringify to convert objects into JSON.

JSON.parse to convert JSON back into an object.

**OOPS**

**Class:** In object-oriented programming, a class is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods).

Example:

class User{

constructor(name){

this.name=name

}

showUser(){

console.log(this.name);

}

}

let user1= new User("megha");

user1.showUser();

let user2 = new User("malay");

user2.showUser();

**Class Expression**

let Product= class{

viewProduct(){

console.log("view product");

}

}

let p1= new Product();

p1.viewProduct();

**Getters/setters**

class User {

constructor(name) {

// invokes the setter

this.name = name;

}

get name() {

return this.\_name;

}

set name(value) {

if (value.length < 4) {

alert("Name is too short.");

return;

}

this.\_name = value;

}

}

let user = new User("John");

alert(user.name); // John

user = new User(""); // Name is too short.

# **Class inheritance**

Class inheritance is a way for one class to extend another class.

Example:

class Child extends Parent{

constructor(a,b,c){

super(a,b);

this.c=c;

}

multiplication(){

let ans= this.a\*this.b\*this.c;

console.log("multiplication="+ans);

}

}

let obj1= new Child(1,2,3);

obj1.add();

obj1.multiplication();

# **Static properties and methods**

class StaticClass{

static objCount(){

console.log("static method run:");

}

}

StaticClass.objCount();

class Article {

static publisher = "Ilya Kantor";

}

console.log(Article.publisher);

**Error handling, "try...catch"**

sometimes our scripts have errors. They may occur because of our mistakes, an unexpected user input, an erroneous server response, and for a thousand other reasons.

**The “try…catch” syntax**

try {

// code...

} catch (err) {

// error handling

}

## [Error object](https://javascript.info/try-catch" \l "error-object)

## When an error occurs, JavaScript generates an object containing the details about it. The object is then passed as an argument to catch:

## name

## Error name. For instance, for an undefined variable that’s "ReferenceError".

## message

## Textual message about error details.

## There are other non-standard properties available in most environments. One of most widely used and supported is:

## stack

## Current call stack: a string with information about the sequence of nested calls that led to the error. Used for debugging purposes.

## Example:

## try{

## console.log(a)

## }

## catch(err){

## console.log(err.message);

## }

## Introduction: callbacks

## A callback is a function passed as an argument to another function

## This technique allows a function to call another function

## A callback function can run after another function has finished

## Example:

## function loadScript(source){

## let scriptTag= document.createElement('script');

## scriptTag.src=source;

## document.head.append(scriptTag);

## }

## loadScript('script.js');

## printData();//error gen while calling this function

## //function not define;

## Callback function will wait until script loaded in file

## Example:

## function loadScript(source,callback){

## let scriptTag= document.createElement('script');

## scriptTag.src=source;

## scriptTag.onload = () => callback(scriptTag);

## document.head.append(scriptTag);

## }

## loadScript('script.js',function(){

## printData();

## });

# **Promise**

"Producing code" is code that can take some time

"Consuming code" is code that must wait for the result

A Promise is a JavaScript object that links producing code and consuming code

A promise object has a state that can be one of the following:

1. Pending
2. Fulfilled with a value
3. Rejected for a reason

## Creating a promise

## const promise = new Promise((resolve, reject) => {

## if (success) {

## resolve(value);

## } else {

## reject(error);

## }

## });

## Consuming a Promise: then, catch, finally

## The then() method accepts two callback functions: onFulfilled and onRejected.

**Example:**

function getUser(){

return new Promise((resolve,reject)=>{

setTimeout(()=>{

resolve([

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

]);

},1000);

});

}

function onFulfilled(users){

console.log(users);

};

function onRejected (err){

console.log(err);

}

const promise = getUser();

promise.then(onFulfilled,onRejected);

2) The catch() method

If you want to get the error only when the state of the promise is rejected, you can use the catch() method of the Promise object:

promise.catch(onRejected);

3) The finally() method

Sometimes, you want to execute the same piece of code whether the promise is fulfilled or rejected.

.finally(() => {

render();

});

**Async/Await**

*"async and await make promises easier to write"*

**async** makes a function return a Promise

**await** makes a function wait for a Promise

example:

async function logJSONData() {

await fetch('https://dummyjson.com/products/')

.then(

res =>res.json()

).then(json=>console.log(json))

.catch(err =>{

console.log("erro message" + err);

})

}

logJSONData();

**Modules, introduction**

As our application grows bigger, we want to split it into multiple files, so called “modules”. A module may contain a class or a library of functions for a specific purpose.

For a long time, JavaScript existed without a language-level module syntax. That wasn’t a problem, because initially scripts were small and simple, so there was no need.

// 📁 sayHi.js

export function sayHi(user) {

alert(`Hello, ${user}!`);

}

// 📁 main.js

import {sayHi} from './sayHi.js';

alert(sayHi); // function...

sayHi('John'); // Hello, John!