JavaScript

.NET CORE

JavaScript (JS) is a programming language that conforms to the ECMAScript specification. JavaScript is high-level, often just-in-time compiled, and multi-paradigm. It has curly-bracket syntax, dynamic typing, prototype-based object-orientation, and first-class functions.

HTTPS://EN.WIKIPEDIA.ORG/WIKI/JAVASCRIPT

Sample .HTML doc

Create a .html and save the below inside

```
<!DOCTYPE html>
<script>
  //your JS code here
</script>
```

Chrome Debugging – Overview

https://javascript.info/debugging-chrome

All modern browsers and most other environments support ***Debugging Tools***.

Debugging Tools is a special UI in ***developer tools*** that makes debugging in the browser much easier. It allows to trace the code step-by-step to see what exactly is going on.

Chrome has many features and most other browsers have a similar process.

Follow this tutorial to learn how to debug in Chrome (or any other browser).

JavaScript – Overview

https://www.w3schools.com/js/default.asp

JavaScript was invented by Brendan Eich in 1995 and became an ECMA standard in 1997. ***ECMA-262*** is the official name of the standard. ***ECMAScript*** is the official name of ***JavaScript***.

It's one of the 3 languages all web developers must learn:

1. ***HTML*** defines the content of web pages.
2. ***CSS*** specifies the layout of web pages.
3. ***JavaScript*** is for programing the behavior of web pages.

JS is well-known as the scripting language for Web pages, but many desktop and server programs use JavaScript also. Node.js, jQuery, Angular, React and many others are examples of programs that use JS or are libraries of JS.

JavaScript – Overview

https://developer.mozilla.org/en-US/docs/Web/JavaScript

JavaScript (JS) is a…

- prototype-based,
- multi-paradigm,
- single-threaded,
- dynamic,
- case-sensitive,
- interpreted (just-in-time compiled)

…programming language that supports

- object-oriented,
- imperative, and
- declarative (e.g. functional programming)

styles.

Do not confuse JavaScript with the Java programming language.
Both "Java" and "JavaScript" are trademarks or registered trademarks of Oracle in the U.S. and other countries.
The two programming languages have very different syntax, semantics, and uses.

JS Versions through time

https://www.w3schools.com/js/js_versions.asp

It is important to understand that JavaScript has changed over time and will continue to change in the future. The major additions to EMCAScript have been:

- with EMCA4 when try/catch handling, better string handling, and numeric output formatting were introduced.
- Then with ES6 and classes, 'let' and 'const', iterators, and arrow functions.
- Then with ES8 and Async Functions.

Since ES7 they decided to release a new version every year with iterative improvements.

ECMAScript Editions

Ver	Official Name	Description
1	ECMAScript 1 (1997)	First Edition.
2	ECMAScript 2 (1998)	Editorial changes only.
3	ECMAScript 3 (1999)	Added Regular Expressions. Added try/catch.
4	ECMAScript 4	Never released.
5	ECMAScript 5 (2009) Read More: JS ES5	Added "strict mode". Added JSON support. Added String.trim(). Added Array.isArray(). Added Array Iteration Methods.
5.1	ECMAScript 5.1 (2011)	Editorial changes.
6	ECMAScript 2015 Read More: JS ES6	Added let and const. Added default parameter values. Added Array.find(). Added Array.findIndex().
7	ECMAScript 2016	Added exponential operator (**). Added Array.prototype.includes.
8	ECMAScript 2017	Added string padding. Added new Object properties. Added Async functions. Added Shared Memory.
9	ECMAScript 2018	Added rest / spread properties. Added Asynchronous iteration. Added Promise.finally(). Additions to RegExp.

Is there an official JS reference?

Nope. We'll use these.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

The Modern JavaScript Tutorial

https://en.wikipedia.org/wiki/JavaScript

JavaScript – Declaring Variables (*var* and *let*)

https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/JavaScript_basics
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Variables#The_difference_between_var_and_let

---

JS creates variables in two ways. '*let* ' and '*var*'

When JavaScript was first created, there was only *var*, which works fine in most cases, but its design can be confusing. If you write a multiline JavaScript program that declares and initializes a variable, you can declare a variable with *var* after you initialize it and, because of hoisting, it will still work. You can also redeclare a variable multiple times with *var*.

let was created to fix issues with *var*.

Use *let* (rather than *var*) unless you need to support versions of IE below v11

```
1  myName = 'Chris';
2
3  function logName() {
4    console.log(myName);
5  }
6
7  logName();
8
9  var myName;
```

```
1  var myName = 'Chris';
2  var myName = 'Bob';
```

JavaScript – Basic Variable Declaration Rules

https://javascript.info/variables#a-variable

- Variables can be considered objects with helper methods. (more)
- You don't have to declare variable types in JavaScript.
- Numbers don't need quotes, but strings and chars do.
- You an declare multiple variables in one line.
- camelCase is conventionally used for variables
- Variables cannot start with a number.
- Variables are case-sensitive.
- Conventionally , Latin chars (0-9, a-z, A-Z) are used for variables.
- Don't use JS keywords.
- Place ***"use strict";*** at the top of .js files to enforce newer conventions (like declaring a variable before defining it).
- Declare an unchanging variable with ***const***.
- Use ALL CAPS for const variables known before compile-time.
- Use meaningful names for variables.
- JS is ***dynamically typed***. This means a variable can be a string and then be a number and then be a float.

```
1  let user = 'John', age = 25, message = 'Hello';
```

```
1  const myBirthday = '18.04.1982';
```

```
1  "use strict";
2
3  num = 5; // error: num is not defined
```

```
1  const COLOR_RED = "#F00";
```

```
2  let message = "hello";
3  message = 123456;
```

JavaScript – Primitive DataTypes

https://javascript.info/types

Datatype	Example	Details
Number (int)	let num = 10;	Operations include *, /, +, -, etc. Includes ***NaN***(not a number) and ***infinity***
Number(floating point)	let num1 = 7087.542	
BigInt	12345678901234567890123456789012345678901234567890n;	Represents any value > 2^53 or < -2^53 (16 digits) or ints of arbitrary length. Use 'n' at the end of a ***BigInt***.
String	let str1 = "there"; let str2 = 'tiger'; let str3 = `Hey ${str1}, ${str2}`;	Must be surrounded by quotes. You can use 'str', "str", or `str` (backticks). Use `str ${otherStr}` for string interpolation. JS has no ***char*** type.
Boolean	let isBool = true; let isTrue = 2>1;	Only has 2 values.
null	let age = null;	A special value which represents "nothing", "empty" or "value unknown". ***null*** is not a reference to an object.
undefined	let x; //x is undefined	The meaning of undefined is "value is not assigned".

JavaScript – Object Data Type and Misc.

https://javascript.info/types

Datatype	Example	Details

Object	//use a constructor
	let john = new User();
	// build an object
	let user = {
	  name: "John",
	  age: 30
	};
		Objects are used to store collections of data and more complex entities in a key-value pair format.

typeof operator	Console.log(typeof x); Console.log(typeof(x));	Returns a string of the type of the argument. It's useful when processing different types differently.
Symbol	let id = Symbol("id");	Object property keys may only be either of string type, or of symbol type. Symbols are guaranteed to be unique.

Operands and Operators

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/A_first_splash
https://javascript.info/operators#terms-unary-binary-operand

An ***operand*** is what ***operators*** are applied to.

For instance, in 5 * 2 there are two operands: the left operand is 5 and the right operand is 2.

JavaScript ***operators*** allow us to perform tests, do math, join strings together, and other such things. A ***unary*** operator has a single operand (let x = 4;), while a ***binary*** operator has two operands (let x = y + z;).

You can also use the + operator to join text strings together (in programming, this is called ***concatenation***). Try entering the following lines, one at a time in the browser console.

If one operand it a string, the other is converted to a string, too. Try ***let new = "hello" + 4;***

Operator	Name	Example
`+`	Addition	`6 + 9`
`-`	Subtraction	`20 - 15`
`*`	Multiplication	`3 * 7`
`/`	Division	`10 / 5`

```
1  let name = 'Bingo';
2  name;
3  let hello = ' says hello!';
4  hello;
5  let greeting = name + hello;
6  greeting;
```

Operator precedence

https://javascript.info/operators#operator-precedence
https://developer.mozilla.org/en/JavaScript/Reference/operators/operator_precedence

Execution order is defined by operator precedence. Parentheses override any precedence. (1 + 2) * 2 = 6.

There are many operators in JavaScript. Every operator has a corresponding precedence number. The one with the larger number executes first. If the precedence is the same, the execution order is from left to right.

You can chain assignments. In ***a = b = c = 2 + 2;*** a, b, and c == 4.

Precedence	Name	Sign
...	...	...
17	unary plus	+
17	unary negation	-
15	multiplication	*
15	division	/
13	addition	+
13	subtraction	-
...	...	...
3	assignment	=

More Operators

https://javascript.info/operators

Operator	Example	Description
%	6%4 == 2	'***%***' is ***modulus*** and gives the remainder.
++	a = 5, a++ == 6	'**++**' increments by 1. '**- -**' decrements by 1. Placed before the variable, '**++**' or '**- -**' occurs before the action. Placed after the variable, '**++**' or '**- -**' happens after the action.
- -	a = 5, a- - == 4	
**	a == 4, b == 3; ***a**b == 48***.	'*******' is the exponent operator. ***a*** is multiplied by itself ***b*** times.
+=	let n = 2; n += 5 == 7	Modify-in-place. Shorthand notation to add, subtract, multiply or divide then save the result to the ***left-hand variable;***
-=	let n = 2; n -= 5 == -3	
*=	let n = 2; n *= 5 == 10	
/=	let n = 10; n /= 5 == 2	

JS –=, ==, and === Operators

https://javascript.info/object#copying-by-reference

=	==	===
Assignment	Equality	Strict equality
Let a = { }; Let c = { }; let b = a; Let d = “134”; Let e = 134	a == b == true a == c == false d == e == true	a === b == true a === c == false d === e == false

Operator	Name	Example

`===`	Strict equality (is it exactly the same?)	`5 === 2 + 4 // false`
		`'Chris' === 'Bob' // false`
		`5 === 2 + 3 // true`
		`2 === '2' // false; number versus string`

`!==`	Non-equality (is it not the same?)	`5 !== 2 + 4 // true`
		`'Chris' !== 'Bob' // true`
		`5 !== 2 + 3 // false`
		`2 !== '2' // true; number versus string`

`<`	Less than	`6 < 10 // true` `20 < 10 // false`
`>`	Greater than	`6 > 10 // false` `20 > 10  // true`

Truthy/Falsy

https://javascript.info/logical-operators
https://developer.mozilla.org/en-US/docs/Glossary/Truthy
https://developer.mozilla.org/en-US/docs/Glossary/Falsy

```
if (true)
if ({})
if ([])
if (42)
if ("0")
if ("false")
if (new Date())
if (-42)
if (12n)
if (3.14)
if (-3.14)
if (Infinity)
if (-Infinity)
```

In JavaScript, a ***truthy*** value is a value that is considered ***true*** when viewed in a ***Boolean*** context. All values are ***truthy*** unless they are defined as ***falsy***.

`false`	The keyword false
`0`	The number zero
`-0`	The number negative zero
`0n`	BigInt, when used as a boolean, follows the same rule as a Number. `0n` is *falsy*.
`""`	Empty string value
null	null - the absence of any value
undefined	undefined - the primitive value
NaN	NaN - not a number

A ***falsy*** value is a value that is considered false when viewed in a Boolean context.

```
if (false)
if (null)
if (undefined)
if (0)
if (-0)
if (0n)
if (NaN)
if ("")
```

JavaScript – Math Helper Functions

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

JavaScript has a built-in Math object which contains a small library of mathematical functions and constants.

Function	Description	Example
Math.random();	returns a floating-point, pseudo-random number in the range 0 to less than 1	Math.random()*10; //3.229976827519583
Math.abs(x);	returns the absolute value of a number	Math.abs(-10 – 6.3); //16.3
Math.pow(x,y);	returns x to the power of y	Math.pow(7, 3); //343
Math.floor(x);	returns the largest integer less than or equal to a given number	Math.floor(5.05) //5
Math.ceiling(x);	rounds a number up to the next largest whole number or integer.	Math.ceil(11.324); //12
Math.max(a,b,…z)	returns the largest of zero or more numbers	Math.max(1, 3, 2); //3

Map and Weak Map

https://javascript.info/map-set#map
https://javascript.info/weakmap-weakset#weakmap

Map	WeakMap
Map is a collection of ***key-value*** data items, just like an ***Object***.	***WeakMap keys*** must be objects, not primitive values:
Any type of key is possible. Even ***Object!***	Weak Map does not prevent ***garbage-collection*** of key objects
Insertion order is used for iteration order.	If you remove all other references to an ***object key***, the ***object*** is removed from memory (and the ***Map()***!).
	WeakMap does not support iteration or methods ***keys(), values(), entries()***
	There is no way to get all ***keys*** or ***values*** from a ***map***
	When working with an object and storing data associated with it (that should only exist while the object is alive) ***WeakMap*** is exactly what's needed.

Set and Weak Set

https://javascript.info/weakmap-weakset#weakset
https://javascript.info/map-set#set

Set	Weak Set
A ***Set*** is a special type collection – "set of ***values***" (without ***keys***), where each ***value*** may occur only once.	Just like ***Set*** but only ***Objects*** are allowed.
The alternative to a ***Set*** is an ***array*** of users and code to check for duplicate users.	An object exists in the set while it is reachable from somewhere else.
	Being "weak", it serves as an additional storage. But only for "yes/no" facts. (use ***.has(obj)*** helper function).
	WeakSet is not iterable and does not support ***.size()***, or ***.keys()***

JavaScript – Function Declarations

https://javascript.info/function-basics

The ***function*** keyword goes first, then goes the name of the function, then a list of parameters between the parentheses (comma-separated) then “the function body”, between { }.

Functions can declare local variables and access variables within the same scope. A local variable declared with the same name ***shadows*** an outer variable. Variables are passed by value in JS.

A function with multiple ***parameters*** can be called with fewer ***arguments***. The unused parameters are shown as ***undefined***.

A ***parameter*** can be given a default value.

```
1  function showMessage(from, text = "no text given") {
2    alert( from + ": " + text );
3  }
4
5  showMessage("Ann"); // Ann: no text given
```

```
1  function showMessage(from, text) { // arguments: from,
2    alert(from + ': ' + text);
3  }
4
5  showMessage('Ann', 'Hello!'); // Ann: Hello! (*)
6  showMessage('Ann', "What's up?"); // Ann: What's up? (**)
```

JavaScript – Functions

https://javascript.info/function-basics

A function can return a value at any point using ***return;***. It can also ***return;*** without a value. Never place return data on a separate line. JS assumes a ***;*** after ***return***.

```
function checkAge(age) {
  if (age >= 18) {
    return true;
  } else {
    return confirm('Do you have permission from your parents?');
  }
}

let age = prompt('How old are you?', 18);

if ( checkAge(age) ) {
  alert( 'Access granted' );
} else {
  alert( 'Access denied' );
}
```

```
function showMovie(age) {
  if ( !checkAge(age) ) {
    return;
  }

  alert( "Showing you the movie" ); // (*)
  // ...
}
```

JavaScript – Function Expressions

https://javascript.info/function-expressions

In JavaScript, a function is a value, so we can deal with it as a value. This code shows a ***function expression***.
We can call it like sayHi(), but it's still a value so we can pass it similar to other kinds of values. A Function Expression is created when the execution reaches it and is usable only from that moment.

```
let sayHi = function() {
  alert( "Hello" );
};
```

Here, the *Function Declaration*:
(1) creates the function and puts it into a variable: sayHi.
(2) copies it into the variable func.
Now the function can be called as both sayHi() and func().
*If there were parentheses after sayHi, func = sayHi() would write the result of the call sayHi() into func, not the function sayHi itself.

```
function sayHi() {   // (1) create
  alert( "Hello" );
}

let func = sayHi;    // (2) copy

func(); // Hello     // (3) run the copy (it works)!
sayHi(); // Hello    //     this still works too (why wouldn't it)
```

Arrow Functions

https://javascript.info/arrow-functions-basics

Arrow Functions are (yet another) a very simple and concise syntax for creating functions. Both the below expressions create a function that accepts arguments *arg1..argN*, then evaluates the expression and returns its result into ***func***.

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

Is the same as...

```
let func = (arg1, arg2, ...argN) => expression
```

```
let double = n => n * 2;
```

```
let sayHi = () => alert("Hello!");
```

This function accepts two arguments: a, b.
It returns the result of ***a + b***.

```
let sum = (a, b) => a + b;

/* This arrow function is a shorter form of:

let sum = function(a, b) {
  return a + b;
};
*/

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

With one argument, ( ) are not required. With zero arguments empty ( ) are required.

```
let sum = (a, b) => {  // the curly brace opens a multiline function
  let result = a + b;
  return result; // if we use curly braces, then we need an explicit "return"
};

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

Arrow Functions vs Lambda Functions

JavaScript – Callback Functions

https://javascript.info/function-expressions#callback-functions

Pass functions as values. (Line 15)The arguments showOk() and showCancel() of the call to ask() are called ***callback functions***.
A function can be passed to be "called back" later (if necessary).
showOk() becomes the callback for a "yes" answer, and showCancel() for a "no" answer.

```
1  function ask(question, yes, no) {
2    if (confirm(question)) yes()
3    else no();
4  }
5
6  function showOk() {
7    alert( "You agreed." );
8  }
9
10 function showCancel() {
11   alert( "You canceled the execution." );
12 }
13
14 // usage: functions showOk, showCancel are passed as arguments to ask
15 ask("Do you agree?", showOk, showCancel);
```

We can use ***Function Expressions*** to write the same function, but much shorter. These are called ***Anonymous Functions*** and are very common

```
1  function ask(question, yes, no) {
2    if (confirm(question)) yes()
3    else no();
4  }
5
6  ask(
7    "Do you agree?",
8    function() { alert("You agreed."); },
9    function() { alert("You canceled the execution."); }
10 );
```

IIFE
Immediately Invoked Function Expression

https://developer.mozilla.org/en-US/docs/Glossary/IIFE
https://en.wikipedia.org/wiki/Immediately_invoked_function_expression

An **IIFE** ***(Immediately Invoked Function Expression)*** (pronounced “iffy”) is a JavaScript function that runs as soon as it is defined. It’s also known as a Self-Executing Anonymous Function

IIFE functions contain two major parts:

- The first is the anonymous function with lexical scope enclosed within the Grouping Operator (). This prevents accessing variables within the IIFE idiom as well as polluting the global scope.
- The second part creates the immediately invoked function expression () through which the JavaScript engine will directly interpret the function.

```
1  (function () {
2      statements
3  })();
```

```
1  (function() {
2      alert("I am not an IIFE yet!");
3  });
```

```
1  // Variation 1
2  (function() {
3      alert("I am an IIFE!");
4  }());
```

IIFE
Immediately Invoked Function Expression

https://developer.mozilla.org/en-US/docs/Glossary/IIFE
https://en.wikipedia.org/wiki/Immediately_invoked_function_expression

```
1 (function () {
2     var aName = "Barry";
3 })();
4 // Variable aName is not accessible from the outside scope
5 aName // throws "Uncaught ReferenceError: aName is not defined"
```

Any variable declared within the expression can not be accessed from outside it.

Assigning an ***IIFE*** to a variable stores the function's return value, not the function definition itself.

```
1 var result = (function () {
2     var name = "Barry";
3     return name;
4 })();
5 // Immediately creates the output:
6 result; // "Barry"
```

Scope with Nested Functions (and Closure)

https://javascript.info/closure

var is function scoped and ***let*** is block scoped. It can be said that a variable declared with ***var*** is defined throughout the program as compared to ***let.***

If a variable is declared inside a code block {...}, it's only visible inside that block.

A nested function can access variables declared inside it's code block and inside it's parent code block.

A nested function can be returned (as a property of a new object or as a result by itself). It can then be used anywhere else and it will still have access to the same outer variables.

```
function sayHiBye(firstName, lastName) {

  // helper nested function to use below
  function getFullName() {
    return firstName + " " + lastName;
  }

  alert( "Hello, " + getFullName() );
  alert( "Bye, " + getFullName() );

}
```

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```

Scope and Closure

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

A ***closure*** is the combination of a ***function*** enclosed with references to its surrounding state (the ***lexical environment***). A ***closure*** gives you access to an outer ***function's*** scope from an inner ***function***.

init() creates local variable (name) and a function (displayName()). displayName() is an inner function ***defined*** inside init(). It's available only within the body of init(). displayName() has no local variables. Because inner functions have access to outer function variables, displayName() accesses ***name*** declared in the parent function, init(). This is Lexical Scoping.

```
1  function init() {
2    var name = 'Mozilla'; // name is a local variable created by init
3    function displayName() { // displayName() is the inner function, a closure
4      alert(name); // use variable declared in the parent function
5    }
6    displayName();
7  }
8  init();
```

Scope and Closure Example

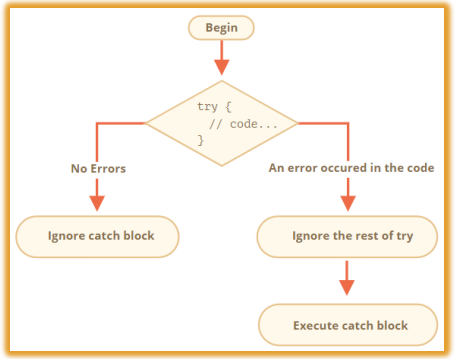
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

makeAdder(x) takes a single argument, x, and ***returns*** a new function. The returned function takes a single argument y, and returns x + y. ***add5*** and ***add10*** are both ***closures***. They share the same function body definition, but store different lexical environments. ***add5***'s lexical environment, x is 5, while in ***add10***, x is 10.

```
1  function makeAdder(x) {
2    return function(y) {
3      return x + y;
4    };
5  }
6
7  var add5 = makeAdder(5);
8  var add10 = makeAdder(10);
9
10 console.log(add5(2));  // 7
11 console.log(add10(2)); // 12
```

Try/Catch/Finally

https://javascript.info/try-catch#the-try-catch-syntax




```
try {

  alert('Start of try runs');  // (1) <--

  lalala; // error, variable is not defined!

  alert('End of try (never reached)');  // (2)

} catch(err) {

  alert(`Error has occurred!`); // (3) <--

}
```

The JS ***Try/Catch*** block works similarly to the C# ***Try/Catch*** Block. There is only one error object generated. The error object has three parts

- Name – the Error Name, Like “Reference Error”.
- Message – a textual message about error details
- Stack – a string with information about the sequence of nested calls that led to the error.

You can create your own error

JavaScript has many built-in constructors for standard errors: ***Error***, ***SyntaxError***, ***ReferenceError***, ***TypeError*** and others.

The ***Finally*** Block always executes.

```
let error = new Error(message);
// or
let error = new SyntaxError(message);
let error = new ReferenceError(message);
// ...
```

```
let json = '{ "age": 30 }'; // incomplete data

try {

  let user = JSON.parse(json); // <-- no errors

  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name"); // (*)
  }

  alert( user.name );

} catch(e) {
  alert( "JSON Error: " + e.message ); // JSON Error: Incomplete data
}
```

JSON and JSON Methods

https://javascript.info/json

JSON (JavaScript Object Notation) is a general format to represent values and objects. Initially it was made for JavaScript, but many other languages have libraries to handle it as well. JSON is used for data exchange. JavaScript provides two methods:

- JSON.stringify to convert objects into JSON.
- JSON.parse to convert JSON back into an object.

The method ***JSON.stringify(student)*** takes the object and converts it into a string.
The ***json*** string is called a JSON-encoded or ***serialized*** or ***stringified*** object. It is ready to be sent over the wire or put into a plain data store. Pay close attention to the format.

```
let student = {
  name: 'John',
  age: 30,
  isAdmin: false,
  courses: ['html', 'css', 'js'],
  wife: null
};

let json = JSON.stringify(student);

alert(typeof json); // we've got a string!

alert(json);
/* JSON-encoded object:
{
  "name": "John",
  "age": 30,
  "isAdmin": false,
  "courses": ["html", "css", "js"],
  "wife": null
}
*/
```

JSON.parse

https://javascript.info/json#json-parse

JSON.parse decodes a ***JSON*** string.

The ***JSON*** may be as complex as necessary, objects and arrays can include other objects and arrays. But they must obey the same ***JSON*** format.

```
5    let userData = '{ "name": "John", "age": 35,
6    "isAdmin": false, "friends": [0,1,2,3] }';
7
8    let user = JSON.parse(userData);
9
10   alert( user.friends[1] ); // 1
```

JavaScript – Type Conversion

Most of the time, operators and functions automatically convert the values given to them to the right type. The three most widely used type conversions are to ***string***, to ***number***, and to ***boolean***.

String(x)	Number(x)		Boolean(x)	
	If input is…	Result is….	If input is…	Result is….
String(value)	undefined	NaN	0 null undefined NaN “” ‘‘	false
Any value can be converted to a string.	null	0		
	true / false	1 / 0		
	string	Whitespaces are ignored. An empty ***string*** becomes 0. An error gives ***NaN***.		
			anything else	true

User Interaction – alert, prompt, confirm

https://javascript.info/alert-prompt-confirm

The browser functions ***alert()***, ***prompt()*** and ***confirm()*** allow interaction with and input from the user.

alert(message)	prompt(title, [default])	confirm()
This shows a message and pauses script execution until the user presses “OK”.	Shows a ***modal*** window with a text message, an input field for the visitor, and the buttons OK/Cancel. Default is the initial value for the input field.	The function confirm shows a ***modal*** window with a question and two buttons: OK and Cancel.