**Modern JavaScript Bootcamp – Andrew Mead**

Angular and React to create web applications

Node JS to create Server applications

To use visual studio, use code.visualstudio.com. others are atom, sublime, webstorm

In vscode, install sublime text keymap, Babel ES6/ES7

Nodejs.org to install server to run javascript

Cmder.net is a terminal emulator for windows

Create a folder, write code in vscode, run it in terminal as – node program.js

Semi column; after a statement is optional in JS

‘let’ is used to define a variable.

Variable name cannot start with a number. It cannot include an exclamation mark!

Equal comparison is ===

Not equal comparison is !==

If (condition) { statements } else if (condition) {statements} else {statement}

If (condition) {if (condition) {statements}}

JS uses lexical scoping (static coping)

Global scope – defined outside of all the code block

Local scope – defined inside the code block

Local scope can access all the variables defined above its scope.

Local scope has predominance over global scope if same variable name is defined in both place, when accessed from local scope. (variable shadowing)

----------

console.log("Hello World")

let fName = "Jaison"

let lName = 'Jacob'

let noName = '' // null and '' points to false

// if else if

if (fName === 'Jaison')

{

console.log("Emil's faher")

}

else if (fName === 'Emil')

{

console.log("Jaison's son")

}

else

{

console.log("Don't know")

}

// nested if

if(true)

{

if(true)

{

console.log("nested if")

}

}

// Global / local

if (true)

{

let fName = 'Esther' // local variable declaration

console.log(fName)

}

console.log(fName)

output:

Hello World

Emil's faher

nested if

Esther

Jaison

----

// function declaration

let buy = function (number, price)

{

console.log("The amout is as follows:")

return number \* price

}

// function call

let amount = buy(5, 50)

console.log(amount)

/\*

The amount is:

The amout is as follows:

250

\*/

-----

‘undefined’ is used to refer to variables for which a value is not assigned

‘null’ is for intentionally not assigning a value to a variable

Template string is used with ${variable\_name} inside quotes. `Hello ${name}` - backtick

Objects: let myBook = { name: ‘1984’, author: ‘George Orwell’, publisher: ‘Penguin Books’, price: $100}

---

let myBook = {

title: '1984', publisher: 'Penguin Books', price: '$100', year: 1984

}

console.log(myBook)

output:

{

title

publi

price

year:

}

---

Function returning an object

// function returning an object

let func = function ()

{

return {

fName: "Jaiosn",

lName: "Jacob"

}

}

let name = func()

console.log(`My nam is: ${name.fName} ${name.lName}`)

output:

My nam is: Jaiosn Jacob

--

A method is an object property, whose value is a function

// object method

let restaurant = {

totalSeats: 100,

guestCount: 10,

checkAvailability: function(count) {

if (count > (this.totalSeats - this.guestCount))

{

console.log("Booking not available")

}

else console.log("Booking available.", `Grade is: ${this.grde}`)

}

}

restaurant.grde = 'A' // declaring an object property outside the object definition

restaurant.checkAvailability(75)

output:

Booking available. Grade is: A

--

Mozilla development network(MDN) – for javascript documentation

This chapter documents all of JavaScript's standard, built-in objects, including their methods and properties.

The term "global objects" (or standard built-in objects) here is not to be confused with **the global object**. Here, "global objects" refer to **objects in the global scope**.

The **global object** itself can be accessed using the [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) operator in the global scope. In fact, the global scope **consists of** the properties of the global object, including inherited properties, if any.

Other objects in the global scope are either [created by the user script](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#Creating_new_objects) or provided by the host application. The host objects available in browser contexts are documented in the [API reference](https://developer.mozilla.org/en-US/docs/Web/API/Reference).

For more information about the distinction between the [DOM](https://developer.mozilla.org/en-US/docs/DOM/DOM_Reference) and core [JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript), see [JavaScript technologies overview](https://developer.mozilla.org/en-US/docs/Web/JavaScript/JavaScript_technologies_overview).

**Standard objects by category**

Value properties

These global properties return a simple value. They have no properties or methods.

* [Infinity](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Infinity)
* [NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN)
* [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined)
* [globalThis](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/globalThis)

Function properties

These global functions—functions which are called globally, rather than on an object—directly return their results to the caller.

* [eval()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval)
* [uneval()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/uneval)
* [isFinite()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/isFinite)
* [isNaN()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/isNaN)
* [parseFloat()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseFloat)
* [parseInt()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt)
* [encodeURI()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/encodeURI)
* [encodeURIComponent()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent)
* [decodeURI()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/decodeURI)
* [decodeURIComponent()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/decodeURIComponent)
* **Deprecated**
  + [escape()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/escape)
  + [unescape()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/unescape)

Fundamental objects

These are the fundamental, basic objects upon which all other objects are based. This includes general objects, booleans, functions, and symbols.

* [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object)
* [Function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function)
* [Boolean](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean)
* [Symbol](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol)

Error objects

Error objects are a special type of fundamental object. They include the basic [Error](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error) type, as well as several specialized error types.

* [Error](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error)
* [AggregateError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/AggregateError)
* [EvalError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/EvalError)
* [InternalError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/InternalError)
* [RangeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RangeError)
* [ReferenceError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ReferenceError)
* [SyntaxError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError)
* [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError)
* [URIError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/URIError)

Numbers and dates

These are the base objects representing numbers, dates, and mathematical calculations.

* [Number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number)
* [BigInt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt)
* [Math](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math)
* [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date)

Text processing

These objects represent strings and support manipulating them.

* [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)
* [RegExp](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp)

Indexed collections

These objects represent collections of data which are ordered by an index value. This includes (typed) arrays and array-like constructs.

* [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)
* [Int8Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Int8Array)
* [Uint8Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Uint8Array)
* [Uint8ClampedArray](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Uint8ClampedArray)
* [Int16Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Int16Array)
* [Uint16Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Uint16Array)
* [Int32Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Int32Array)
* [Uint32Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Uint32Array)
* [Float32Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Float32Array)
* [Float64Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Float64Array)
* [BigInt64Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt64Array)
* [BigUint64Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigUint64Array)

Keyed collections

These objects represent collections which use keys. The iterable collections ([Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map) and [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set)) contain elements which are easily iterated in the order of insertion.

* [Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map)
* [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set)
* [WeakMap](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap)
* [WeakSet](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakSet)

Structured data

These objects represent and interact with structured data buffers and data coded using JavaScript Object Notation (JSON).

* [ArrayBuffer](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer)
* [SharedArrayBuffer](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer)
* [Atomics](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Atomics)
* [DataView](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/DataView)
* [JSON](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON)

Control abstraction objects

Control abstractions can help to structure code, especially async code (without using deeply nested callbacks, for example).

* [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)
* [Generator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator)
* [GeneratorFunction](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/GeneratorFunction)
* [AsyncFunction](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/AsyncFunction)
* AsyncGenerator
* AsyncGeneratorFunction

Reflection

* [Reflect](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect)
* [Proxy](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy)

Internationalization

Additions to the ECMAScript core for language-sensitive functionalities.

* [Intl](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl)
* [Intl.Collator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Collator)
* [Intl.DateTimeFormat](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/DateTimeFormat)
* [Intl.ListFormat](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ListFormat)
* [Intl.NumberFormat](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NumberFormat)
* [Intl.PluralRules](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/PluralRules)
* [Intl.RelativeTimeFormat](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RelativeTimeFormat)
* [Intl.Locale](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Locale)

WebAssembly

* [WebAssembly](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly)
* [WebAssembly.Module](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Module)
* [WebAssembly.Instance](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Instance)
* [WebAssembly.Memory](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Memory)
* [WebAssembly.Table](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Table)
* [WebAssembly.CompileError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/CompileError)
* [WebAssembly.LinkError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/LinkError)
* [WebAssembly.RuntimeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/RuntimeError)

Other

* [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments)
* --

Constant variables cannot be reassigned – const name = ‘Jaison’

Array helps to store list based data

Declaring variable with ‘var’ did not warn if the variable is re-declared. Var is function scoped, not block scoped.

If the variable is declared with ‘var’ below the code, only the declaration is hoisted to the top of the code.

Array items are created using square brackets [ ]

Array.shift() – removes the first element from the array

Array.unshift(‘item’) – adds to the beginning of the array

Array.splice(index,item-to-remove/replace,item-to-add) – removes/inserts items at specific place.

A callback function is a function that is passed to a function

Array.forEach(argument) – to loop over an array

//Array

const myArray = ["one", "two", "three", "four"]

console.log(myArray)

myArray.shift()

console.log(myArray)

myArray.unshift("One again")

console.log(myArray)

myArray.splice(2,0,"added")

console.log(myArray)

myArray.splice(1,1,"replaced")

console.log(myArray)

myArray.forEach(function(item,index){

console.log(item,":",index)

})

output:

[ 'one', 'two', 'three', 'four' ]

[ 'two', 'three', 'four' ]

[ 'One again', 'two', 'three', 'four' ]

[ 'One again', 'two', 'added', 'three', 'four' ]

[ 'One again', 'replaced', 'added', 'three', 'four' ]

One again : 0

replaced : 1

added : 2

three : 3

four : 4

---

For loop: for(initialize; condition; increment/decrement) // incr / decr happens after running the expression

{

Expression

}

//Array for loop

const myArray = ["one", "two", "three", "four"]

for (let i=0; i<4; i++) {

console.log(myArray[i])

}

output:

one

two

three

four

---

Array.indexOf(‘item’) – returns the index of the item in the array. A -1 means no part in the array.

Two objects are equal if both points to same memory

// Array of objects

const arrayObjects = [ {

title: "First", pulisher: "Penguin"

}, {

title: "Second", publisher: "DC Books"

}, {

title: "Third", publisher: "Green Books"

}]

console.log(arrayObjects)

// finding the index of the object in the array

const note = arrayObjects.findIndex(function(item){

return item.title === "Second"

})

console.log(note)

output:

[

{ title: 'First', pulisher: 'Penguin' },

{ title: 'Second', publisher: 'DC Books' },

{ title: 'Third', publisher: 'Green Books' }

]

1

---

// Array of objects

const arrayObjects = [ {

title: "First", pulisher: "Penguin"

}, {

title: "Second", publisher: "DC Books"

}, {

title: "Third", publisher: "Green Books"

}]

console.log(arrayObjects)

// finding the item at the index of the object in the array

const note = arrayObjects.find(function(item){

return item.title === "Second"

})

console.log(note)

output:

[

{ title: 'First', pulisher: 'Penguin' },

{ title: 'Second', publisher: 'DC Books' },

{ title: 'Third', publisher: 'Green Books' }

]

{ title: 'Second', publisher: 'DC Books' }

--

// Filtering an Array of objects

const arrayObjects = [ {

title: "First", publisher: "Penguin"

}, {

title: "Second", publisher: "DC Books"

}, {

title: "Third", publisher: "Green Books"

}]

console.log(arrayObjects)

const myFilter = arrayObjects.filter(function(obj){

const title1 = obj.title.toLowerCase().includes('z')

const pub1 = obj.publisher.toLowerCase().includes('e')

return title1 || pub1

})

console.log(myFilter)

output:

[

{ title: 'First', publisher: 'Penguin' },

{ title: 'Second', publisher: 'DC Books' },

{ title: 'Third', publisher: 'Green Books' }

]

[

{ title: 'First', publisher: 'Penguin' },

{ title: 'Third', publisher: 'Green Books' }

]

---

// Filtering an Array of

const arrayObjects = [ {

title: "First", publisher: "Penguin"

}, {

title: "Second", publisher: "DC Books"

}, {

title: "Third", publisher: "Green Books"

}]

console.log(arrayObjects)

/\*

const myFilter = arrayObjects.filter(function(obj){

const title1 = obj.title.toLowerCase().includes('z')

const pub1 = obj.publisher.toLowerCase().includes('e')

return title1 || pub1

}) \*/

// Using a function to filter array objects

const myFilter = function(obj,item\_to\_filter){

return obj.filter(function (obj){

const title1 = obj.title.toLowerCase().includes(item\_to\_filter)

const pub1 = obj.publisher.toLowerCase().includes(item\_to\_filter)

return title1 || pub1

})

}

console.log(myFilter(arrayObjects,'books'))

output:

[

{ title: 'First', publisher: 'Penguin' },

{ title: 'Second', publisher: 'DC Books' },

{ title: 'Third', publisher: 'Green Books' }

]

[

{ title: 'Second', publisher: 'DC Books' },

{ title: 'Third', publisher: 'Green Books' }

]

---

// Filtering an Array of

const arrayObjects = [ {

title: "First", publisher: "Penguin"

}, {

title: "Second", publisher: "DC Books"

}, {

title: "Third", publisher: "Green Books"

}]

console.log(arrayObjects)

// Deleting an object from an array of objects by search

const theFunc = function(obj, str){

const index = obj.findIndex(function(obj){

return obj.title.toLowerCase().includes(str)

})

if (index >= -1) {

obj.splice(index,1)

}

//console.log(obj)

return obj

}

console.log(theFunc(arrayObjects,'second'))

output:

[

{ title: 'First', publisher: 'Penguin' },

{ title: 'Second', publisher: 'DC Books' },

{ title: 'Third', publisher: 'Green Books' }

]

[

{ title: 'First', publisher: 'Penguin' },

{ title: 'Third', publisher: 'Green Books' }

]

---

The **filter()** method **creates a new array** with all elements that pass the test implemented by the provided function.

//sort

var numbers = [44, 21, 5, 11, 3];

numbers.sort(function (a, b) {

return a - b;

});

console.log(numbers);

output:

[ 3, 5, 11, 21, 44 ]

---

// Array sorting

const todos = [{text: "order cat food", completed: true},

{ text: "order dog food", completed: true },

{ text: "order bird food", completed: false },

{ text: "order fish food", completed: true },

{ text: "order our food", completed: false },

]

console.log(todos)

const arr = function (todos){

todos.sort(function(a,b){

if(!a.completed && b.completed){

return -1

}

else if (!b.completed && a.completed){

return 1

}

else return 0

})

}

arr(todos)

console.log(todos)

output:

[

{ text: 'order cat food', completed: true },

{ text: 'order dog food', completed: true },

{ text: 'order bird food', completed: false },

{ text: 'order fish food', completed: true },

{ text: 'order our food', completed: false }

]

[

{ text: 'order bird food', completed: false },

{ text: 'order our food', completed: false },

{ text: 'order cat food', completed: true },

{ text: 'order dog food', completed: true },

{ text: 'order fish food', completed: true }

]

// Expense tracker

const account = {

name: 'Andrew Mead',

expenses: [],

income: [],

addExpense: function (description, amount) {

this.expenses.push({

description: description,

amount: amount

})

},

addIncome: function (description, amount) {

this.income.push({

description: description,

amount: amount

})

},

getAccountSummary: function () {

let totalExpenses = 0

let totalIncome = 0

let accountBalance = 0

this.expenses.forEach(function (expense) {

totalExpenses = totalExpenses + expense.amount

})

this.income.forEach(function (income) {

totalIncome = totalIncome + income.amount

})

accountBalance = totalIncome - totalExpenses

return `${this.name} has a balance of $${accountBalance}. $${totalIncome} in income. $${totalExpenses} in expenses.`

}

}

account.addExpense('Rent', 950)

account.addExpense('Coffee', 2)

account.addIncome('Job', 1000)

console.log(account.getAccountSummary())

output:

Andrew Mead has a balance of $48. $1000 in income. $952 in expenses.

---

To install a live server, use command: npm install –g live-server

* 1. Create a folder under the current folder
  2. Create a file named index.html
  3. Got to the parent folder of the folder where the index.html is saved
  4. Run command: live-server folder-name

The html file will open in a browser

App.js

console.log(`JS for notes-app`)

// Adding an element to the document

const b = document.createElement('p') // create

b.textContent = "From Javascript" // add content

document.querySelector("body").appendChild(b) // append to document

// removing an element from the document

const a = document.querySelectorAll('p') // select the element type to remove

a.forEach(function(para){

if(para.textContent.includes("Hello")){ // query for the content

para.remove() // remove

}

})

---

Empty html elements such as input text, text area need NOT be closed </>. But others.

Eventlistener for form is ‘submit’

In form, e.preventDefault() – prevents default behavior of forms – adding data to the URL

To store items local, use CRUD

localStorage.setItem(‘key’,’value’)

localStorage.getItem(‘key’)

localStorage.setItem(‘key’,’value)

localStorage.removeItem(‘key’)

JSON.stringify(object/array) // JSON.parse(stringified object)

const a = {

title: "Mad world", author: "Jesus"

}

const j = JSON.stringify(a)

console.log('stringified: ',j)

localStorage.setItem('book',j)

p = localStorage.getItem('book')

i = JSON.parse(p)

console.log('parsed: ',i)

localStorage.removeItem('book')

console.log(localStorage.getItem('book'), 'item deleted')

output:

stringified: {"title":"Mad world","author":"Jesus"}

parsed:

Object { title: "Mad world", author: "Jesus" }

null item deleted

---

Refactoring is restructuring of code without affecting its external behavior.

Multiple javascript files can be inserted to to the source code of one html file.

Global properties of javascript files can be accessed in other javascript files. But in html file, js files are executed in order.

In html document, the function defining js file should be on top, then the function calling js file.

Debugging code is getting the bugs out of code.

Debugger in code triggers the debugging tool in the browser.

Checkbox.setAttribute(“type”,”checkbox”)

Gitbbub uuid ---- for third party library

<a href”/index.html”> Index page </a> -- this directs to the index page

location.assign("http://sehion.org")

#${id} -- hash

innerHTML is a property of every element. It tells you what is between the starting and ending tags of the element, and it also let you sets the content of the element.

property describes an aspect of an object. It is something an object has as opposed to something an object does.

<p id="myParagraph">

This is my paragraph.

</p>

You can select the paragraph and then change the value of it's innerHTML with the following command:

document.getElementById("myParagraph").innerHTML = "This is my paragraph";

--

Window.innerHeight // window.innerWeight

Window.eventListener()

Moment.js.com is the third party library to use with Date

moment()

// Soring in descend

const a = [1,2,3,4,5,9,3,10,9,11,20,15]

const b = a.sort(function(n,p){

if (n < p) return 1

else if (n === p) return 0

else return -1

})

console.log(b)

output:

[

20, 15, 11, 10, 9,

9, 5, 4, 3, 3,

2, 1

]

---

JavaScript doesn't support functions that return multiple values. However, you can wrap multiple values into an array or an object and return the array or the object. Use destructuring assignment syntax to unpack values from the array, or properties from objects.

Arrow functions:

Const a = () => { statements }

Const a = () => expression

// arguments gives arguments passed on to the function parameter

const a = function(a,b){

console.log(arguments)

}

a(1,2,3,'a','k')

output:

[Arguments] { '0': 1, '1': 2, '2': 3, '3': 'a', '4': 'k' }

---

Arguments only bound in regular functions, not in arrow functions.

‘this.’ Will not work with arrow functions.

// Ternary operator

const age = 27

message = age > 20 ? 'Major': 'Minor'

console.log(message)

output:

Major

---

Truthiness in js: any value that’s not resolve to ‘true’ is false. Null, empty string, undefined, 0, false are all falsy.

Type coercion:

const a = '5' + 5 // concatenates

console.log(a)

const b = '5' - 5 // subtracts

console.log(b)

output:

55

0

---

‘===’ is a strict equality operator. ‘==’ is loose equality. (‘5’ == 5 ) is true. Where as (‘5’ === 5) is false

Typeof(5) will be ‘number’

// try-catch and throw

const a = (v) => {

if (v > 10) return 'Yes'

else

throw Error("Not as expected, error")

}

try {

const k = a(11)

console.log(k)

}

catch(e){

console.log(e.message)

}

output:

Yes

----

//'use strict'

const a = () => {

b = 10 // when b is not defined, under non-use-strict mode, 'b' is made global by Js

return b

}

console.log(a())

---

Functions that are created with the ‘new’ operator are called constructor functions.

The property ‘prototype’ is an object where we put everything we want to share with instances in our constructor function.

// constructor function and prototype property

const Person = function(fName,lName,age){

this.fName = fName

this.lName = lName

this.age = age

}

Person.prototype.getBio = function(){

return `First Name: ${this.fName}, Last Name: ${this.lName}, Age: ${this.age}`

}

const p1 = new Person('Jaison','Jacob',45)

const p2 = new Person('Sonia', 'Jaison', 39)

const n = p1.getBio()

const n1 = p2.getBio()

console.log(n)

console.log(n1)

output:

First Name: Jaison, Last Name: Jacob, Age: 45

First Name: Sonia, Last Name: Jaison, Age: 39

---

String.split(‘<split character>’)

Almost everything is an ‘object’ in javascript.

Object.hasOwnProperty(‘property’) – checks if the object has this property.

Const obj = new Object()

Object.prototype.someMethod = () => { ‘statements’ }

Primitive objects has no value: string, number, Boolean, null, undefined.

String, number, Boolean are all object wrappers also.

Prototypal inheritance:

Object: myObject -> Object.prototype -> null

Array: myArray -> Array.prototype -> Object.prototype -> null

Function: myFunction -> Function.prototype -> Object.prototype -> null

String: myString -> String.prototype -> Object.prototype -> null

Number: myNumber -> Nnumber.prototype -> Object.prototype -> null

Boolean: myBool -> Boolean.prototype -> Object.prototype -> null

A class in JS is an alternate way to create a constructor function with its methods.

class <class\_name> {

constructor(arguments){

Statemetns

}

}

---

Extends:

Class Employee extends Person { statements }

class Person{

constructor(fName,lName,age,likes=[]){

this.fName = fName

this.lName = lName

this.age = age

this.likes = likes

}

}

class Employee extends Person{

constructor(fName, lName, age,position, likes) {

super(fName, lName, age, likes)

this.position = position

}

}

const emp = new Employee('Jaiosn', 'Jaconb', 45, 'Dispatcher',['Sports', 'Trading'])

console.log(emp)

output:

Employee {

fName: 'Jaiosn',

lName: 'Jaconb',

age: 45,

likes: [ 'Sports', 'Trading' ],

position: 'Dispatcher'

}

----

// Getters and setters

const data = {

get location(){

return this.city

},

set location(value){

this.city = value.trim()

}

}

data.location = 'Liyon'

console.log(data)

console.log(`City: ${data.location}`)

output:

{ location: [Getter/Setter], city: 'Liyon' }

City: Liyon

---

Every httpresponse comes with a ‘status’ code.

Httpstatuses.com – to get all the http status code to us

100’s informational

200’s success

300’s redirectional

400’s client error

500’s server error

Headers in a request/response header is API specific

Restcountries.eu – for JSON data

Asynchronous: Can do something a little, can move to some other task, comeback and finish current one.

Synchronous: Finish one task, then move to another.

//HttpRequest

const req = new XMLHttpRequest()

req.addEventListener('readystatechange',(e) => {

if(e.target.readyState === 4){

const data = JSON.parse(e.target.responseText)

console.log(data)

}

})

req.open('GET','http://puzzle.mead.io/puzzle')

req.send()

console.log("querying ..")

output:

querying..

Object { puzzle: "Adventure Story" }

---

// callback function

const getPuzzle = (puzzle) => {

puzzle('heheheh',’success’)

}

getPuzzle((error,puzzle) => {

console.log(puzzle)

})

output:

heheheh

---

// callback function with error argument

const getPuzzle = (puzzle) => {

puzzle(undefined,'success')

// puzzle('error','success')

}

getPuzzle((error,puzzle) => {

if(error) console.log(error)

else console.log(puzzle)

})

utput:

success

---

A closure is a function with the combination of the lexical scope of the function in which it is defined

//closure

const myPrint = () => {

let message = 'Hi'

const myFunc = () => {

message = 'Biden'

console.log(message)

}

return myFunc

}

const a = myPrint()

a()

// output

// Biden

---

// carrying function

const a = (a) => {

return (b) => a + b

}

const b = a(10)

console.log(b(5))

//output: 15

---

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

//call back

const greet = (name) => {

console.log('Hello ' + name);

}

const process = (callback) => {

//var name = prompt('Please enter your name.');

//let name = "Hai "

callback("haiii");

}

process(greet);

// output:

// Hello haiii

---

What is a Promise?

A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it’s not resolved (e.g., a network error occurred). A promise may be in one of 3 possible states: fulfilled, rejected, or pending. Promise users can attach callbacks to handle the fulfilled value or the reason for rejection.

Promises are eager, meaning that a promise will start doing whatever task you give it as soon as the promise constructor is invoked. If you need lazy, check out observables or tasks.

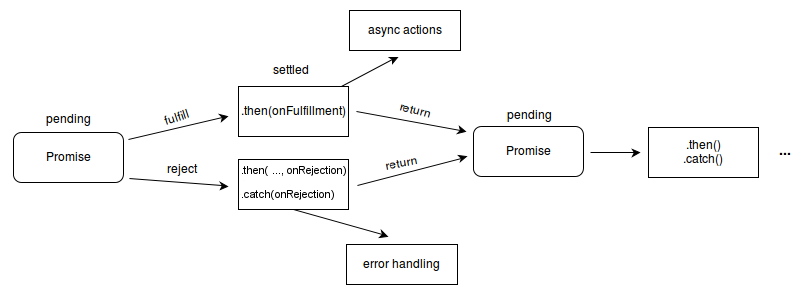
A **Promise** is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a *promise* to supply the value at some point in the future.

A Promise is in one of these states:

* *pending*: initial state, neither fulfilled nor rejected.
* *fulfilled*: meaning that the operation was completed successfully.
* *rejected*: meaning that the operation failed.

A pending promise can either be *fulfilled* with a value or *rejected* with a reason (error). When either of these options happens, the associated handlers queued up by a promise's then method are called. If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.

As the [Promise.prototype.then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) and [Promise.prototype.catch()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch) methods return promises, they can be chained.



setTimeout(,<time delay>) – to set timeout

// promise

const prom = new Promise((resolve, reject) => {

//resolve("Perfect")

reject("Imperfect")

})

prom.then((data) => {

console.log(data)

}, (err) => {

console.log(err)

}

)

//ouptut:

//Imperfect

---

When we return a promise from another promise handler, it’s a promise chaining.

The instance method of the **Promise** object such as then() , catch() , or finally() returns a separate **promise** object. Therefore, you can call the **promise's** instance method on the return **Promise** . The successively calling methods in this way is referred to as the **promise chaining**.

// promse chaining

let p = new Promise((resolve, reject) => {

setTimeout(() => {

resolve(10);

}, 3 \* 100);

});

p.then((result) => {

console.log(result); // 10

return result \* 2;

}).then((result) => {

console.log(result); // 20

return result \* 3;

}).then((result) => {

console.log(result); // 60

return result \* 4;

});

Output:

10

20

60

---

Fetch API is a new way to make HTTPRequest

Fetch(url,{}) – returns a promise

//fetch - is a promise

// run this in live-server

fetch('http://puzzle.mead.io/puzzle',{}).then((response) =>{

if (response.status == 200){

return response.json()

} else {

throw new Error('Data error')

}

}).then((data)=>{

console.log(data.puzzle)

}).then((error)=>{

console.log(error)

})

---

Fetch API has promise built-in

Async function, await operator

‘undefined’ is the default return value from a function if nothing is explicitly returned.

Pointplace