Javascript Manual

A Javascript Manual for absolute beginners.

Made by: Ali Sumbal

Contents

Data Types ____ ***3***
VAR vs LET ____ ***3***
Additional Operators in JS ____ ***3***
If-Else and Switch statements ____ ***4***
FOR and WHILE Loops ____ ***5***
Math object cheat sheet ____ ***5***
Number constants ____ 5
Generating Random Numbers ____ 5
Rounding methods ____ 5
Arithmetic and calculus methods ____ 6
String cheat sheet ____ ***6***
TypeOf Operator ____ ***6***
Arrays ____ ***7***
Objects ____ ***7***
i. Dot Notation: ____ 8
ii. Bracket Notation: ____ 8
Arrays are Objects ____ **9**
Object Methods ____ 10
Bugs VS Errors ____ ***11***
TRY-CATCH Block (Error handling) ____ ***12***
Error Prevention using Error-Handling Example ____ 14
Defensive Programming Example ____ 14
Functions ____ ***15***
The Functional Programming paradigm ____ ***15***
First-class functions ____ 15
Higher-order functions ____ 16
Pure functions and side-effects ____ 16
OOP ____ ***16***
Classes ____ **17**
Principles of OOP ____ **17**
Inheritance ____ 17
Encapsulation ____ 18
Abstraction ____ 18
Polymorphism ____ 18
Default Parameters ____ **19**
FOR OF LOOPS and objects ____ ***20***
Template literals ____ ***20***
Data Structures ____ ***21***
- Objects: ____ 21
- Arrays: ____ 21
- Maps: ____ 22

- Sets: 22

Spread and Rest Operator 23

Using Spread/Rest operator To: 24

Join arrays, objects 24

Add new members to Array without using `push()` 25

Convert a string to an array 25

Copy either an object or an array into a separate one 25

JavaScript in the Browser 26

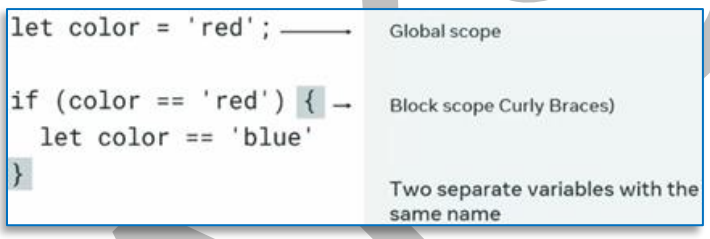
JSON (JavaScript Object Notation) 27

Data Types

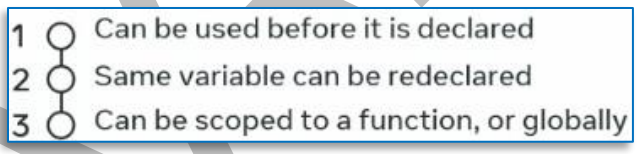
There are seven primitive data types in JavaScript.

1) **String**
2) **Number**
3) **Boolean** (TRUE/FALSE)
4) **Null** (used when variable has no value / intentional absence of object or variable's value)
   **can also be the return value of some built-in JS methods.*
5) **Undefined** (used when value is not assigned yet, but should be assigned later)
6) **BigInt** (for large numbers)
7) **Symbol** (unique identifier)

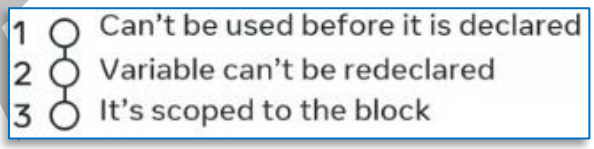
VAR vs LET



❖ **VAR**



❖ **LET**



Additional Operators in JS

Following are some additional operators in JS:

- Modulus operator: `%` : returns the remainder of division
- Equality operator: `==` : checks if two values are equal.
- Strict equality operator: `===`: compares for both the values and the data types.

- Inequality operator: `!=`
- Strict inequality operator: `!==`
- Addition assignment operator: `+=`

If-Else and Switch statements

- **If-else statements:**

```
var age= 10;
if(age >=65)
    console.log("You get your income from your pension");
else if(age< 65 && age>=18)
    console.log( "Each month you get a salary");
else if(age< 18)
    console.log("You get an allowance");
else
    console.log("The value of the age variable is not numerical");
```

- **Switch statements:**

```
var day= "Friday";
switch(day)
{
    case "Monday":
        console.log("1");
        break;
    case "Tuesday": ···
    case "Wednesday": ···
    case "Thursday": ···
    case "Friday": ···
    case "Saturday": ···
    case "Sunday":
        console.log("7");
        break;
    default:
        console.log("There is no such day");
        break;
}
```

FOR and WHILE Loops

- **FOR loop:**

```
for(var i=1; i<= 5; i++)
{
    console.log(i);
}
```

- **WHILE loop:**

```
var i= 1;
while(i<= 5){
    console.log(i);
    i++;
}
```

Math object cheat sheet

Number constants

Here are some of the built-in number constants that exist on the Math object:

- The PI number: `Math.PI`
- The Euler's constant: `Math.E`
- The natural logarithm of 2: `Math.LN2`

Generating Random Numbers

- `Math.random()` - generate a decimal number between 0 and 0.99.

Rounding methods

These include:

- `Math.ceil()` - rounds up to the closest integer
- `Math.floor()` - rounds down to the closest integer
- `Math.round()` - rounds up to the closest integer if the decimal is `.5` or above; otherwise, rounds down to the closest integer
- `Math.trunc()` - trims the decimal, leaving only the integer

Arithmetic and calculus methods

Here is a non-conclusive list of some common arithmetic and calculus methods that exist on the `Math` object:

- `Math.pow(2,3)` - calculates the number `2` to the power of `3`, the result is `8`
- `Math.sqrt(16)` - calculates the square root of `16`, the result is `4`
- `Math.cbrt(8)` - finds the cube root of `8`, the result is `2`
- `Math.abs(-10)` - returns the absolute value, the result is `10`
- Logarithmic methods: `Math.log()`, `Math.log2()`, `Math.log10()`
- Return the minimum and maximum values of all the inputs: `Math.min(9,8,7)` returns `7`, `Math.max(9,8,7)` returns `9`.
- Trigonometric methods: `Math.sin()`, `Math.cos()`, `Math.tan()`, etc.

String cheat sheet

List of all the methods covered in this cheat sheet:

- `charAt()`
- `concat()`
- `indexOf()`
- `lastIndexOf()`
- `split()`
- `toUpperCase()`
- `toLowerCase()`

TypeOf Operator

The type of operator accepts and evaluates a parameter and returns the name of the data type represented as a string.

- **Syntax**: `typeof(Parameter);`

Arrays

```
1   //function that takes an array as input and display all items of this array
2   function listArrayItems(arr) {
3       for (var i = 0; i < arr.length; i++) {
4           console.log(i, arr[i])
5       }
6   }
7   var colors = ['red', 'orange', 'yellow', 'green', 'blue', 'purple', 'pink'];
8   listArrayItems(colors);
```

Declaring an Array

Objects

One of the most common ways of building an object in JavaScript is using the object literal syntax: `{}`.

```
var user = {}; //create an object
```

An entire object can be immediately built, using the **object literal syntax**, by specifying the object's properties, delimited as key-value pairs. It essentially consists of two steps:

1) Declaring a new variable and assigning an object literal to it - in other words, this: `var assistantManager = {}`
2) Assigning the values to each of the object's keys, using the assignment operator, `:`

```
//creating an object with properties and their values
var assistantManager = {
    rangeTilesPerTurn: 3,
    socialSkills: 30,
    streetSmarts: 30,
    health: 40,
    specialAbility: "young and ambitious",
    greeting: "Let's make some money"
}
```

Property value

socialSkills = 50;

Property key

i. Dot Notation:

An alternative approach of building objects is to first save an empty object literal to a variable, then use the **Dot Notation** to declare new properties on the fly, and use the assignment operator to add values to those properties.

```
var house2 = {};
house2.rooms = 4;
house2.color = "pink";
house2.priceUSD = 12345;
```

We can also combine the two approaches. For example:

```
var house = {
    rooms: 3,
    color: "brown",
    priceUSD: 10000,
}
console.log(house); // {rooms: 3, color: "brown", priceUSD: 10000}
house.windows = 10;
console.log(house); // {rooms: 3, color: "brown", priceUSD: 10000, windows: 10}
```

This means we can also **update** already existing properties, not just **add** new ones:

```
house.windows = 11;
console.log(house); // {rooms: 3, color: "brown", priceUSD: 10000, windows: 11}
```

ii. Bracket Notation:

An alternative syntax to the Dot notation is *the **Brackets Notation***. In the following example we code **`house2`** object using Bracket Notation:

```
var house2 = {};
house2["rooms"] = 4;
house2['color']= "pink";
house2["priceUSD"] = 12345;
console.log(house2); // {rooms: 4, color: 'pink', priceUSD: 12345}
```

Using the Brackets notation, we just wrap each property's key **as a string** (hence we can use numbers also as property keys), inside the single or double quotes, and then wrap the entire property key into an opening and a closing square bracket.

With the brackets notation, we can **also add space** characters inside property names, like this:

```
car["number of doors"] = 4;
console.log(car); // {color: 'green', speed: 100, number of doors: 5}
```

Another useful thing that Bracket notation has but is not available in the Dot notation: It **can evaluate expressions**. This can be explained using the following example:

```
var arrOfKeys = ['speed', 'altitude', 'color'];
var drone = {
    speed: 100,
    altitude: 200,
    color: "red"
}
for (var i = 0; i < arrOfKeys.length; i++) {
    console.log(drone[arrOfKeys[i]])
}
```

Output

```
100
200
red
```


Arrays are Objects

JavaScript treats arrays as objects. This means arrays also have some built-in properties and methods. One of the most commonly used built-in methods on arrays are the `push()` and the `pop()` methods.

To add new items to an array, I can use the `push()` method:

```
var fruits = [];
fruits.push("apple"); // ['apple']
fruits.push('pear'); // ['apple', 'pear']
```

To remove the last item from an array, I can use the `pop()` method:

```
fruits.pop();
console.log(fruits); // ['apple']
```

Using these we can build a function that takes all its arguments and pushes them into an array, like this:

```
function arrayBuilder(one, two, three) {
    var arr = [];
    arr.push(one);
    arr.push(two);
    arr.push(three);
    return arr;
}
var simpleArr = arrayBuilder('apple', 'pear', 'plum');
console.log(simpleArr); // ['apple','pear','plum']
```

Object Methods

An object consists of key-value pairs, known as properties. Properties can take many data types, including functions. If a function is a property of an object, it is then referred to as a method. This function can be accessed only through the JavaScript object that it is a member of. For example, the "**log**" method, which belongs to the "**console**" object, can only be accessed through the **console** object.

```
console.log('Hello world');
```

```
var car = {};

car.color = "red";

//add a method to the car object so that it can be called as car.turnkey()
car.turnKey = function() {
  console.log('engine running');
}
console.log(car);
car.turnTheKey();
```

Calling "turnTheKey" method

```
{
  mileage: 98765,
  color: 'red',
  turnTheKey: [Function (anonymous)],
}
The engine is running
```

Output

Bugs VS Errors

- **Bug:** It causes a program to run in an unintended way.

```
function addNums(a, b) {
  console.log(a + b);
}

addNums("1", 2);

console.log("Still running");
```

```
Console
12
Still running
```

- **Error:** A faulty piece of code that prevents the program from further execution (causes a program to stop running)

JavaScript reports the error by outputting an error message to the console. Some of the most common types of Error in JavaScript are:

I. **Syntax Error:** occurs when you write a piece of code that JavaScript cannot read.

```
var word = "hello;
               |
          Syntax error
```

```
Console
SyntaxError: Invalid or unexpected token
```

II. **Type Error:** occurs when you use a feature/functionality, but not as it was intended to be used for/with.

```
"hello".pop() // Uncaught TypeError: "hello".pop is not a function
```

III. **Reference Error:** occurs when a variable is not defined, but you attempt to use it in your code.

```
console.log(c + d);

console.log('This line never runs');
```

```
Console
ReferenceError: c is not defined
```

Does not execute

IV. **Range Error**: occurs when we give a value to a function, but that value is out of the allowed range of acceptable input values.
Example:-
Let's convert an everyday Base 10 number to a Base 2 number (i.e binary number).

```
(10).toString(2); // '1010'
```

However, if we try to use a non-existing number system, such as an imaginary Base 100, we will get the Range Error, because a non-existing Base 100 system is *out of range* of the number systems that are available to the `toString()` method:

```
(10).toString(100); // Uncaught RangeError
```

There are some other errors in JavaScript. These other errors include:

- Aggregate Error
- Internal Error
- URI Error

TRY-CATCH Block (Error handling)

The *Try-Catch* statement keeps a program running, even when it encounters an error. The statements uses the key words *"throw", "try" and "catch"* to work with errors in JavaScript and prevent your code from stopping. This process is more commonly known as error handling.

The *Try-Catch* statement has 2 blocks, *try-block* and *catch-block.* If a piece of code throws an error, we can wrap it inside a *Try* block. Then we can catch the error with the *Catch* block, and use it to do something, like outputting the error message to the console. We use the "*throw"* keyword in the *try-block*, to force an error to be thrown from the *try block* to the *catch block.*(Remember, we can use the throw keyword outside the try block, but it will not be possible to catch it)*

The *catch block* accepts a variable as a parameter, which is an object. This is the actual **error** that is thrown from the *try block*. We can name it anything we like, but it's best to keep it short and meaningful. In this case we named it *"err"*.

```
try {

 throw new Error();

} catch(err) {

  console.log(err)

}

console.log('This line now runs')
```

```
Console
ReferenceError
This line now runs
```

Example#1

```
try {
    console.log(a + b)
} catch(err) {
    console.log(err)
    console.log('There was an error')
    console.log('The error was saved in the error log')
}
console.log("My program does not stop")
```

```
ReferenceError: a is not defined
    at Object.<anonymous>
    (C:\Users\ajdik\AppData\Local\Temp\t
    CodeRunnerFile.javascript:2:17)
    at Module._compile (node:internal/
    modules/cjs/loader:1103:14)
    at Object.Module._extensions..js
    (node:internal/modules/cjs/
    loader:1157:10)
    at Module.load (node:internal/module
    cjs/loader:981:32)
    at Function.Module._load
    (node:internal/modules/cjs/
    loader:822:12)
    at Function.executeUserEntryPoint [a
    runMain] (node:internal/modules/
    run_main:77:12)
    at node:internal/main/
    run_main_module:17:47
There was an error
The error was saved in the error log
My program does not stop
```

Example#2: Manually Throwing an Error

```
try {
    throw new ReferenceError();
} catch(err) {
    console.log(err)
    console.log('There was a Reference Error')
}
console.log("My program does not stop")
```

```
ReferenceError
    at Object.<anonymous>
    (C:\Users\ajdik\AppData\Local\Temp\temp
    CodeRunnerFile.javascript:2:11)
    at Module._compile (node:internal/
    modules/cjs/loader:1103:14)
    at Object.Module._extensions..js
    (node:internal/modules/cjs/
    loader:1157:10)
    at Module.load (node:internal/modules/
    cjs/loader:981:32)
    at Function.Module._load
    (node:internal/modules/cjs/
    loader:822:12)
    at Function.executeUserEntryPoint [as
    runMain] (node:internal/modules/
    run_main:77:12)
    at node:internal/main/
    run_main_module:17:47
There was a Reference Error
My program does not stop
```

Error Prevention using Error-Handling Example

```
function addTwoNums(a, b) {
    try{
        if(typeof(a) != "number") {
            throw new ReferenceError("the first argument is not a number");
        }
        else if(typeof(b) != "number") {
            throw new ReferenceError("the second argument is not a number");
        }
        else {
            console.log(a + b);
        }
    }
    catch(err){
        console.log("Error!", err);
    }
}
addTwoNums(5, "5");
console.log("It still works");
```

Output

```
Error! ReferenceError: the second argument is not a number
    at addTwoNums (file:///tmp/dhxcz/submission.mjs:15:19)
    at file:///tmp/dhxcz/submission.mjs:25:1
    at ModuleJob.run (internal/modules/esm/module_job.js:183:25)
    at async Loader.import (internal/modules/esm/loader.js:178:24)
    at async Object.loadESM (internal/process/esm_loader.js:68:5)
It still works
```

Defensive Programming Example

Defensive programming is all about assuming that all the arguments a function will receive are of the wrong type, the wrong value or both.

In other words, you are assuming that things will go wrong and you are proactive in thinking about such scenarios before they happen, so as to make your function less likely to cause errors because of faulty inputs.

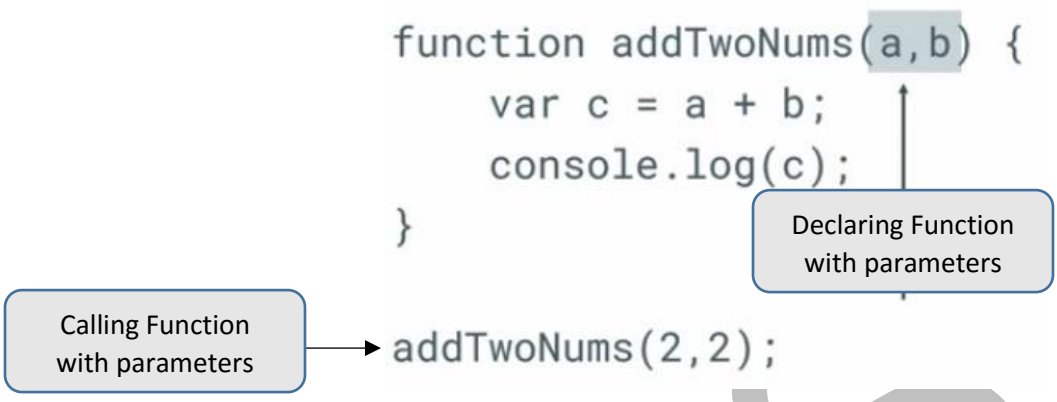
```
function letterFinder(word, match) {
    var condition1 = typeof(word) == "string" && word.length >= 2;
    var condition2 = typeof(match)== "string" && match.length == 1;
    if(condition1 == true && condition2 == true) {
        for(var i = 0; i < word.length; i++) {
            if(word[i] == match) {
                console.log('Found the', match, 'at', i)
            } else {
                console.log('---No match found at', i)
            }
        }
    } else {
        console.log("Please pass correct arguments to the function.");
    }
}

letterFinder("hello", "h");
```

Output

```
Found the h at 0
---No match found at 1
---No match found at 2
---No match found at 3
---No match found at 4
```

Functions

Declaring and Calling a Function:



The Functional Programming paradigm

"The Functional Programming paradigm works by keeping the data and functionality separate. Its counterpart, OOP, works by keeping the data and functionality grouped in meaningful objects."

There are many more concepts and ideas in functional programming. Here are some of the most important ones:

- **First-class functions**
- **Higher-order function**
- **Pure functions and side-effects**

First-class functions

It is often said that functions in JavaScript are "first-class citizens". What does that mean? It means that a function in JavaScript is just another value that we can:

- pass to other functions
- save in a variable
- return from other functions

In other words, a function in JavaScript is just a value - from this vantage point, almost no different then a string or a number.

Higher-order functions

A higher-order function is a function that has either one or both of the following characteristics:

- It accepts other functions as arguments
- It returns functions when invoked

This is simply a feature of the language. The language itself allows me to pass a function to another function, or to return a function from another function.

Pure functions and side-effects

A pure function returns the exact same result as long as it's given the same values.

```
function addTwoNums(a, b) {
console.log(a + b)
}
```

This function will always return the same output, based on the input. For example, as long as we give it a specific value, say, a **5**, and a **6**:

```
addTwoNums(5,6); // 11
```

Another rule for a function to be considered pure is that it should not have side-effects. A side-effect is any instance where a function makes a change outside of itself. This includes:

- changing variable values outside of the function itself, or even relying on outside variables
- calling a Browser API (even the console itself!)
- calling **Math.random()** - since the value cannot be reliably repeated

OOP

“A paradigm in which we organize our programs using objects to group related data and functionality.”

OOP helps developers to mimic the relationship between objects in the real world.

- Allows to write modular code,
- Makes code more flexible
- Makes code reusable.

*(The *"**this**" keyword* is an alias for the name of the object. It refers to the object itself without specifying the object's name)

Classes

- A class is built using the “class” keyword.
- The *constructor* function **assigns values**, whether hardcoded or passed-in parameters, to the object’s properties at the time of object instantiation. It sets up the mechanism for saving instance-specific values (data). When we `console.log(this)`, the object gets returned only with the *properties (data)* that was build using the constructor() function.
- While *console logging* `Object.getPrototypeOf(this)` results in only shared *methods* being stored on the prototype.
- In classes, `super` is used to specify what property gets inherited from the super-class in the sub-class.
- Instance of the class is created using the keyword “new” and that class name, followed by opening and closing parentheses, and optional arguments, based on how the class itself is defined.

“In conclusion, the *class* syntax in JS allows us to clearly separate **individual object's data** - which exists on the object instance itself - from the **shared object's functionality (methods)**, which exist on the prototype and are shared by all object instances.**”**

```
class Car {
    constructor(color, speed) {
        this.color = color;
        this.speed = speed;
    }

    turboOn() {
        console.log("turbo is on!")
    }
}
```

```
const car1 = new Car("red", 120)
```

Principles of OOP

Four fundamental OOP principles are *inheritance, encapsulation, abstraction* and *polymorphism.*

Inheritance

To setup the inheritance relation between classes in JavaScript, we use the `extends` keyword, as in `class B extends A`.

```
class Animal { /* ...class code here... */ }
class Bird extends Animal { /* ...class code here... */ }
class Eagle extends Bird { /* ...class code here... */ }
```

Encapsulation

making a code implementation "hidden" from other users, in the sense that they don't have to know how my code works in order to "consume" the code.

Abstraction

Data abstraction refers to providing only essential information about the data to the outside world, while hiding the background details or implementation.

Reference: Similarly in C++ we use Abstraction using Access Specifiers (public, private, protected) on Class members

Polymorphism

To understand what polymorphism is about, let's consider some real-life objects.

- A door has a bell. It could be said that the bell is a property of the door object. This bell can be rung. When would someone ring a bell on the door? Obviously, to get someone to show up at the door.
- Now consider a bell on a bicycle. A bicycle has a bell. It could be said that the bell is a property of the bicycle object. This bell could also be rung. However, the reason, the intention, and the result of somebody ringing the bell on a bicycle is not the same as ringing the bell on a door.

```
const bicycle = {
    bell: function() {
        return "Ring, ring! Watch out, please!"
    }
}
const door = {
    bell: function() {
        return "Ring, ring! Come here, please!"
    }
}
bicycle.bell(); // "Get away, please"
door.bell(); // "Come here, please"
```

At this point, one can conclude that methods having the exact same name can have opposite intent, based on what object it is used for.

Now, to make this code truly polymorphic, we will add another function declaration:

```
function ringTheBell(thing) {
    console.log(thing.bell())
}
ringTheBell(bicycle); // Ring, ring! Watch out, please!
ringTheBell(door); // "Ring, ring! Come here, please!"
```

Thus, the exact same function produces different results, based on the context in which it is used.

However, at the same time, you can override some parts of the shared functionality or even the complete functionality, in some other parts of the OOP structure.

```
class Bird {
    useWings() {
        console.log("Flying!")
    }
}
class Eagle extends Bird {
    useWings() {
        super.useWings()
        console.log("Barely flapping!")
    }
}
class Penguin extends Bird {
    useWings() {
        console.log("Diving!")
    }
}
var baldEagle = new Eagle();
var kingPenguin = new Penguin();
baldEagle.useWings(); // "Flying! Barely flapping!"
kingPenguin.useWings(); // "Diving!"
```

Default Parameters

Default parameter inside a function definition, goes hand in hand with the defensive coding approach. Default parameters allow us to build a function that will run with default argument values even if we don't pass it any arguments, while still being flexible enough

to allow us to pass custom argument values and deal with them accordingly. Additionally, this approach really shines when building inheritance hierarchies using classes, as it makes it possible to provide only the custom properties in the sub-class, while still accepting all the default parameters from the super-class constructor.

FOR OF LOOPS and objects

*Following built-in methods take objects as parameters.

`Object.keys()` **method:** returns array of properties

`Object.values()` **method:** returns array of object's values

`Object.entries()` **method:** returns an array listing both the keys and the values.

Template literals

An alternative way of working with strings was introduced in the ES6 addition to the JS language. Up until ES6, the only way to build strings in JS was to delimit them in either single quotes (' ') or double quotes (" ").Alongside the previous ways to build strings, ES6 introduced the use of backtick (`) characters as delimiters:

```
`Hello, World!`
```

Template string:

- Allows for **variable interpolation**

```
let greet = "Hello";
let place = "World";
console.log(`${greet} ${place} !`) //display both variables using template literals
```

- Can span multiple lines.
- Allows for **expression evaluation**.

```
//it's possible to perform arithmetic operation inside a template literal expression
console.log(`${1 + 1 + 1 + 1 + 1} stars!`)
```

Data Structures

When having a coding task, we can think of the Data Structures that can be used. This helps in determining how the data can be represented in our program, which will further help us to code the solution. Most common examples of Data Structures in JS are:

- **Objects:** are unordered, non-iterateable collection of key-value pairs.

  Object's **Properties can be converted to Array** using the following method:

```
const result = [];
const drone = {
    speed: 100,
    color: 'yellow'
}
const droneKeys = Object.keys(drone);//variable is assigned keys of 'drone' object
droneKeys.forEach( function(key) {  //each key is iterated over.
    result.push(key, drone[key])   //key-value pair is pushed into 'result' array
})
console.log(result)
```

```
const result = [];
const drone = {
    speed: 100,
    color: 'yellow'
}
function arrayMaker(key) {
    result.push(key, drone[key])
}
const droneKeys = Object.keys(drone);
droneKeys.forEach(arrayMaker);
console.log(result)
```

- **Arrays:** Three specific methods that exist on arrays:

  1) **`forEach`:** method accepts **a function that will work on each array item**. The function's first parameter is the current array item itself, and the second (optional) parameter is the index.

```
const fruits = ['kiwi','mango','apple','pear'];
function appendIndex(fruit, index) {
    console.log(`${index}. ${fruit}`)
}
fruits.forEach(appendIndex);
```

or

```
const veggies = ['onion', 'garlic', 'potato'];
veggies.forEach( function(veggie, index) {
    console.log(`${index}. ${fruit}`);
});
```

  2) **`filter`:** makes the function iterate over each array element, and **filters the array based on a specific test**. The array items that pass the test are returned.

```
const nums = [0,10,20,30,40,50];
function test(num) {
    return num > 20;
}
console.log(nums.filter(test));
```

or

```
const nums = [0,10,20,30,40,50];
let values= nums.filter( function(num) {
    return num > 20;
})
console.log(values);
```

Output: `[ 30, 40, 50 ]`

3) **`map`:** used to map each array item over to another array's item, based on whatever work is performed inside the function that is passed-in to the map as a parameter.

```
let array1= [0,10,20,30,40,50];
function mapper(num) {
    return num / 10;
}
let array2= array1.map(mapper);
console.log(array1);
console.log(array2);
```

```
let array1= [0,10,20,30,40,50];
let array2= array1.map(function(num) {
    return num / 10
})
console.log(array1);
console.log(array2);
```

- **Maps:** are iterateable.

  To make a new Map, we can use the **`Map`** constructor:

```
let bestBoxers = new Map();
```

  To set a specific value, you need to use the **`Set()`** method

```
bestBoxers.set(1, "The Champion");
bestBoxers.set(2, "The Runner-up");
bestBoxers.set(3, "The third place");

console.log(bestBoxers);
```

  To get a specific value, you need to use the **`get()`** method

```
bestBoxers.get(1); // 'The Champion'
```

- **Sets:** is a collection of unique values. We can use the **`Set`** constructor to build a new set. The **`Set`** constructor can accept an array. This means that we can use it to quickly filter an array for unique members.

```
const repetitiveFruits = ['apple','pear','apple','pear','plum', 'apple'];
const uniqueFruits = new Set(repetitiveFruits);
console.log(uniqueFruits);
```

Spread and Rest Operator

Spread:

Spread is used to unpack a box, for example, to unpack an array. Spread operator is characterized by three dots **(...)**. Spread operator allows you to pass all array elements into a function without having to type them all individually.

```
let top7 = ['1','2','3','4','5','6','7']

function print(a,b,c,d) {
    console.log(a,b,c,d);
}
print(...top7);
```

```
"C:\User
tempCode
1 2 3 4

[Done] e
seconds
```

Rest:

The rest operator, on the other hand, is used to build a smaller box, and pack items into it. The rest operator can be used to de-structure existing array items, rather than typing them out again. It allows you to take items from an array and use them to create a separate sub-array. The rest operator, therefore, gives us what is left over of the source array, as a separate sub array.

```
let top7 = ['1','2','3','4','5','6','7']

const [first, second, third, ...secondVisit] = top7
console.log(first,second,third);
console.log(secondVisit);
```

```
[Running] node
"C:\Users\toshiba\AppDa
tempCodeRunnerFile.java
1 2 3
[ '4', '5', '6', '7' ]
```

The rest operator is also useful in functions too. The Rest symbol along with a variable name (…varName), can be used as parameter in a function. This variable acts as an array, so we can use array methods on this parameter. It's important to know, that the rest parameter, must be the last parameter in the function definition. This means, that adding any other parameter after the rest operator would result in an error.

```
function addTaxToPrices(taxRate, ...itemsBought) {
    return itemsBought.map(item => taxRate * item)
}

let shoppingCart = addTaxToPrices(1.1,46,89,35,79);

console.log(shoppingCart); // [50.6, 97.9, 38.5, 86.9]
```

```
function count(...basket) {
    console.log(basket.length)
}

count(10, 9, 8, 7, 6);
```

A mix of Spread and Rest

```
let top7 = ['1','2','3','4','5','6','7']
function print(a, ...arry) {
    console.log(a);
    console.log("Remaining:",...arry )

    if(arry.length==0) {
        return
    }
    print(...arry);
}

print(...top7);
```

```
[Running] node "C:\User
1
Remaining: 2 3 4 5 6 7
2
Remaining: 3 4 5 6 7
3
Remaining: 4 5 6 7
4
Remaining: 5 6 7
5
Remaining: 6 7
6
Remaining: 7
7
Remaining:
```

Using Spread/Rest operator To:

Join arrays, objects

- Joining Arrays:

```
const fruits = ['apple', 'pear', 'plum']
const berries = ['blueberry', 'strawberry']
const fruitsAndBerries = [...fruits, ...berries] // concatenate
console.log(fruitsAndBerries); // outputs a single array
```

- Joining Objects:

```
const flying = { wings: 2 }
const car = { wheels: 4 }
const flyingCar = {...flying, ...car}
console.log(flyingCar) // {wings: 2, wheels: 4}
```

Add new members to Array without using `push()`

```
let veggies = ['onion', 'parsley'];
veggies = [...veggies, 'carrot', 'beetroot'];
console.log(veggies);
```

Convert a string to an array

```
const greeting = "Hello";
const arrayOfChars = [...greeting];
console.log(arrayOfChars); //  ['H', 'e', 'l', 'l', 'o']
```

Copy either an object or an array into a separate one

- Copying an object into a separate one

```
const car1 = {
    speed: 200,
    color: 'yellow'
}
const car 2 = {...car1}

car1.speed = 201

console.log(car1.speed, car2.speed)
```

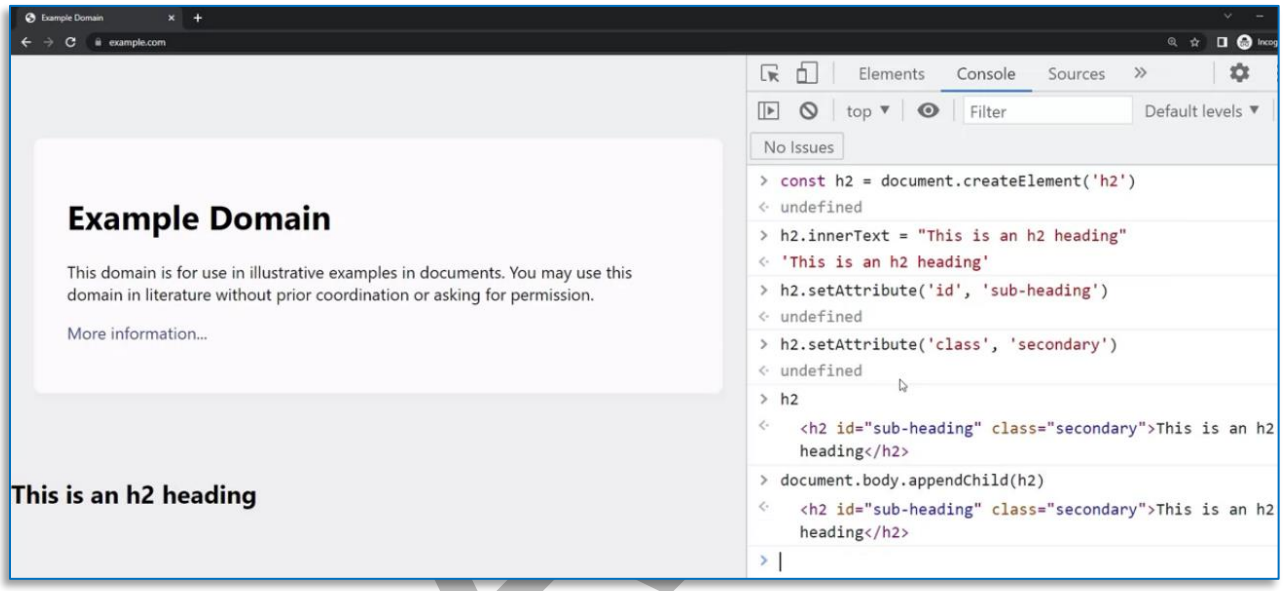
- Copying an Array into a separate one

```
const fruits1 = ['apples', 'pears']
const fruits2 = [...fruits]
fruits1.pop()
console.log(fruits1, "not", fruits2)
```

JavaScript in the Browser

The webpage stored in the browser's memory is known as the Document Object Model (DOM). The document object which you can access by typing the keyword ‘document’ returns this stored webpage. We use selectors in JavaScript to quickly find specific objects in a document. For doing this, the JavaScript selectors work with the ‘document’ object.

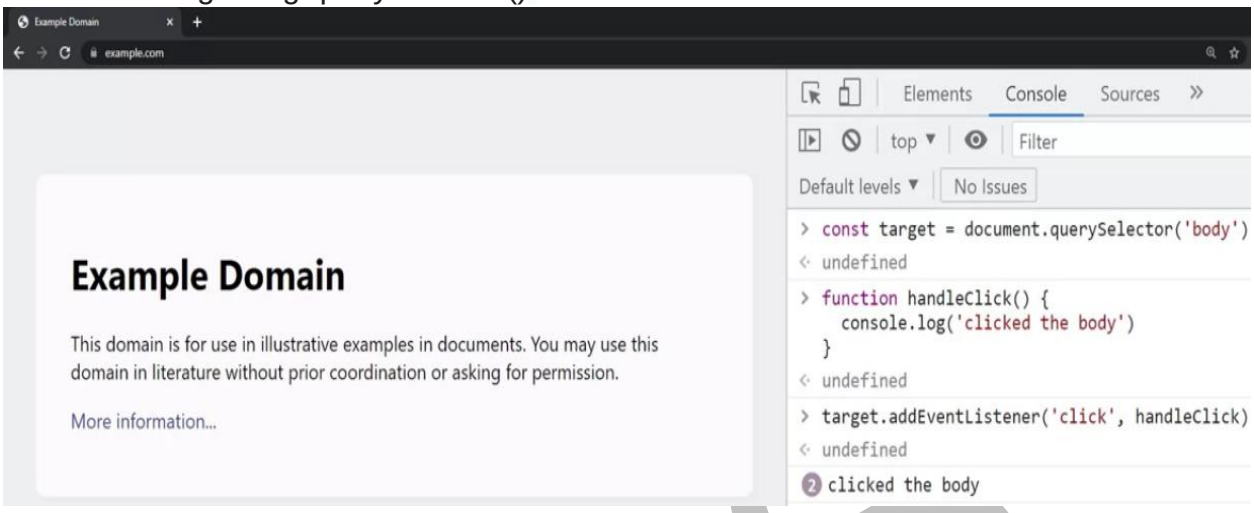
Using JavaScript DOM manipulation to create a ‘h2’ (heading-2) element:



JavaScript Selectors:

- querySelector(): This method returns the first element in the document that matches the selector.
- querySelectorAll(): This method returns all elements in the document that match the selector.
- getElementById(): This method returns a specific element whose id matches the selector.
- getElementsByClassName(): This method returns all elements in the document that has the class specified in the selector.

Event handling using querySelector():



JSON (JavaScript Object Notation)

JSON is a data interchange format based on JavaScript objects.

Before JSON, the most common data interchange file format was XML (Extensible Markup Language). However XML's syntax required more characters to describe the data that was sent. Also, since it was a specific stand-alone language, it wasn't as easily inter-operable with JavaScript.

JSON is a string, but it must be a properly-formatted string. In other words, it must adhere to some rules. If a JSON string is not properly formatted, JavaScript would not be able to parse it into a JavaScript object. Only a subset of values in JavaScript can be properly stringified to JSON and parsed from a JavaScript object into a JSON string. These values include:

- **primitive values:** strings, numbers, bolleans, null
- **complex values:** objects and arrays (no functions!)
- Objects have double-quoted strings for all keys
- Properties are comma-delimited both in JSON objects and in JSON arrays, just like in regular JavaScript code

You can convert a JSON string to a JavaScript object so that you can work with that object's properties. To do this you need to use the global built-in JSON object and its 'parse' method `JSON.parse(jsonStrg)`. You can also convert a JavaScript Object to

a JSON string using the JSON object and its ‘stringify’ method `JSON.stringify(jsObj)`