Javascript Manual

Roadmap

0. Introduction
1. Statements
2. Values and datatypes
3. Expressions
4. Variables
5. Functions
6. Flow Control Structures
7. Arrays
8. Objects

0: Introduction

- *Javascript* is a programming language.
- It can run in the browser (front-end) or in the server (back-end), with *NodeJS*.
- It's similar to PHP, C# and Java, but it's *less strict* and *dynamically typed*.

Programming principles

- A program has two moments:
    - First, it's *written* by a programmer (or autor); the program code is stored in a file like `script.js`
    - Second, it's *executed* by an interpreter (like Chrome); the program code is executed statement by statement.

1: Statements

A **computer program** is made of **statements**.

Each **statement** is an instruction to the computer

```
// Example: this program has 3 statements
const myVariable = 'hello'; // first statement
console.log('hello there'); // second statement
alert('hello again'); // third statement
```

2: Values and datatypes

> *A **value** is a piece of data, or information.*

Examples: the number `7`, the string `"hello"`, the boolean value `true`, a date.

In Javascript, values have also a **data type**, which tell us *what is the range of options that a value can be* and *what operations can be done with them*.

- `string`: it can be a list of alphanumeric characters: letters,numbers, symbols, like `h`, `hello`, or `hello 🌎 world`. A common operation is *concatenation* (i.e.: `"hello" + "world" === "helloworld"`.
- `number`: it can be any integer or real number (`1`, `0`, `-5`, `7.8`, etc.), and it supports arithmetical operations (sum, rest, division).
- `boolean`: it can be only two values: `true` or `false`. Operations are generally of `boolean` logic (`AND`, `OR`, `NOT`)
- `object`: it represents *composite* data structures, made of many combined values. They also have a reference to a model or blueprint, which can be a **Prototype** or a **Class**.
- `undefined`: it can be only the value `undefined`, used when the value is missing (for example, in an uninitialized variable)

Note 1: The operator `typeof` tells us the data type of a value.

```
typeof "hello"; // "string"
typeof 7.88; // "number"
typeof true; // "boolean"
typeof new Date(); // "object"
typeof undefined; // "undefined"
```

Note 2: values can be tranformed from a data type into other, this is called ***casting***.

```
console.log('5' + 1); // result: "51". The first value is a string,
not a number
// solution, cast the string into a number
console.log( Number('5') + 1 ); // result: 6
```

3: Expressions

> *An **expression** is any valid unit of code that resolves to a **value**.*

```
// Example
const a = 1; // we assign the value 1 to the variable a;
const b = 5 + 3 - 7;
//        \_________/ <---- this is the expression
// first, 5 + 3 is calculated into 8
// then, 8 - 7 is calculated into 1
// there's no more to calculate, so 1 is assigned to variable b
```

Three rules of expressions:

> *1: They're made of **operators** and **values***
>
> *2: When they appear in the code, they are **resolved**: that is, they're **evaluated** (or calculated) into a **value**.*
>
> *Corollary:*
> *3: Wherever you can use a **value**, you can also use an **expression***

```
// Example of rule 3:
console.log(7); // we send 7 to the console
console.log( 6 + 1 ); // we send 7 to the console
console.log( someVariable + 8); // we send the result of adding the
content of someVariable and 8 to the console
```

Operators

They indicate transformations (calculations) over one value, or between two values.

They are evaluated from left to right, but they have precedence rules (i.e: multiplication before addition).

Parenthesis `()` can be used to override precedence.

```
1 + 2 + 5; // 1 plus 2 equals 3; Then, 3 plus five equals 8. Result:
8.
1 + 2 * 4; // First, 2 times 4 gives 8; Then, 1 plus 8 gives 9.
Result: 9.
(1 + 2) * 4; // First, 1 plus 2 fives 3; then, 3 times 4 gives 12.
Result: 12.
```

There are many types of operations, they usually use symbols (`+`, `/`, `&&`) but also keyword (`typeof`, `in`).

Classified by:

- Unary, binary or tertiary: how many values they operate on:
    - Unary: `!`. ie: `!false`
    - Binary: the most common. i.e: `1 + 2`.
    - Tertiary: `true ? 1 : 2`.
- Type:
    - arithmetic (`+`, `*`, etc.),
    - comparison (`>=`, `==`, `===`, etc.),
    - logical (`&&`, `||`, `!`)
    - string (`+`, in the context of strings, means *concatenation*)

- Others...

We can also consider as operations three special cases:

- **Referencing** a variable or function (see below in Variables and Functions). This is finding an identifier (a variable) and obtaining the **value** that is stored into that variable.
- **Invoking** a function: taking a function and executing it (passing it certain arguments).
- **Accessing** a member: taking an object and obtaining one part of it.

4: Variables

> *Variables are containers for storing data (values).*

Variables are useful to store data in them, then later referencing that data. **They're useful when we don't know what specific data we will have in our program, but we know what the data represents**.

```
// Basic syntax
var a = 1;              // this is a shortcut for declaring and
assigning at once
 ^   ^ ^ ^---- Value
 |   | |------ Assignment operator
 |   |-------- Variable name (identifier)
 |-----------  Keyword var
```

There are only 3 things we can do with variables

1. **Declaring** a variable. `var myVariable`. We define a **identifier** that will be available later to be assigned, re-assigned or referenced. This can be done only once per variable.
2. **Assigning** a value to a variable. `myVariable = 2`. We store a value into the variable. This can be done more than once (unless it's a constant).
3. **Referencing** a variable. `const anotherVariable = myVariable + 1` or `console.log(myVariable)`. When we use the name of the variable in an **expression**, we are taking the container and extracting what is stored inside it.

Declaration keywords

- `var` is classic JS, but not recommended anymore. It has different scoping rules.
- `let` is recommended way to declare variables.
- `const` declares constants, variables that can be assigned only once

Scope

An important concept is **scope**: when a variable is **declared**, it is accessible only **in the block where it's declared**.

```
const a = 1;    // declared in the top scope
console.log(a); // prints 1
{
  const b = 2;  // declared in the inner scope
  console.log(b); // prints 2
  console.log(a); // prints 1
}
// back into top scope
console.log(a); // prints 1 again
console.log(b) // This will throw "Error: b is not defined."
```

In the previous example, there are two **scopes**: the top one (outside the block) and the next (inside the block).

- When we reference an **identifier**, Javascript searches for it, first in the current scope (where we are doing the reference). Then, in the parent scope. Then, in the parent of the parent, etc.
- Said otherwise: the inner scope can reference the outer scope, but not viceversa.
- A variable that is defined in an scope can be re-defined in the inner scope: this is called **shadowing** (but not recommended).

5: Functions

> ***Functions*** *are reusable "subprograms" that can be* ***invoked*** *to be executed. They can take* ***arguments*** *and* ***return values****.*

Functions can be provided by Javascript itself or the runtime environment (like `console.log` or `alert`) or we can create our own functions.

There are **two** things we can do with functions.

1. **Invoking** (aka running, calling, executing) a function.

```
// Basic syntax of invoking a function
alert("hello");
  ^  ^-------^------ parenthesis indicate that we want to invoke the
function
  |      ^---------- one optional argument (a value)
  ------------------ the name of the function (an identifier)
```

This statement finds a function named `alert` (which is an **identifier**), and **invokes it** passing one argument (the string `"hello"`). The function `alert` (provided by the execution environment, in this case, the browser) runs the function and executes its internal code.

2. **Defining** (aka writing, authoring) a function

We can define our own functions if we want. We'll do 3 examples to explain 3 types of functions

1: Functions as subprograms

```
// Example 1: simple function with statements
//  |------------------------- keyword "function" indicates we are
defining a variable
//  ↓        ↓----------------- we must give a name to the function,
used later to invoke it
```

```
//                          ↓--- Between the {} we declare the "body"
of the function
function sayHelloFriends() {
  console.log('hello!'); // This is the first statement
  console.log('friends!'); // This is the second statement
}

// Once we have defined our own function, we can invoke it now (or
later)

sayHelloFriend();
// Result (in the console):
// hello!
// friends!
```

2: Functions that return values

Functions **always** return values by using the keyword `return` in their body. `return` can be omitted, which means that the function will return `undefined`. The return value will be used as the result of invoking the function in an expression.

```
// Simple function that always returns 1
function alwaysOne() {
  return 1; // When invoked, this function will always return one
}

alwaysOne(); // Nothing will happen. The value 1 is returned but not
used.
const someNumber = alwaysOne(); // The result of invoking the
function is 1; 1 is stored in the variable someNumber
const otherNumber = 3 + alwaysOne() + 7; // The invocation is part of
an expression

console.log(someNumber); // 1
console.log(otherNumber); // 11
```

```
// function without a return statement
function getNothing() {
  alert('hello');
  alert('bye');
}

const result = getNothing(); // this will trigger 2 alerts
console.log(result); // undefined
```

3: Parameters and arguments

Functions can be declared with **parameters**: variables that can be used in the body, and that will be assigned values when being invoked. The values that will be passed are called **arguments**.

We have to think in two ways:

- when *writing a function*, we think in generic terms, in **parameters**. We don't see a value; we see a variable whose value that can be anything.
- when *calling a function*, we think in concrete terms. We pass values as **arguments** to the function.

```
// let's invoke a function, thinking concretely
const result = sum(3, 4); // Calling function sum with 2 arguments: 3
and 4
// The arguments are concrete values. they can also be expressions
(because "wherever we can use a value we can use an expression")
const result = sum(someNumber, 4); // someNumber is referenced and
transfomed into a value before being passed as argument to sum

// let's write a function, thinking generically, abstractly

//            |-----|------ we have declared two parameters
//            |     |       they are like variables, available only
inside the function body
//            ↓     ↓       concrete values will be assigned WHEN the
function is invoked
function sum(num1, num2) {
```

```
  console.log('My first number is', num1);  // Parameters are
variables: they can be referenced
  console.log('My second number is', num2); // We don't know what
will be sent to console when
                                            // writing the function,
only when invoking it
  const result = num1 + num2;               // simple operation
referencing two variables
  return result;
}


const three = sum(1, 2);  // The arguments are 1 and 2
console.log(three);       // 3
const seven = sum(three, 4);
console.log(seven);       // 7

const oneTwoThreeFour = sum( sum(1, 2), sum(3, 4) );
console.log(oneTwoThreeFour); // 10
```

When they tell you:

> *Write a function that **takes** or **receives** one thing and **returns** another thing*

You'll know that your function definition will use **parameters** to specify what it will **receive**, and **return** to give back

6: Flow Control Structures

Statements in a program are executed one after another: the flow is *lineal*. There are other structures that allow us to change the flow and make it diverge or loop.

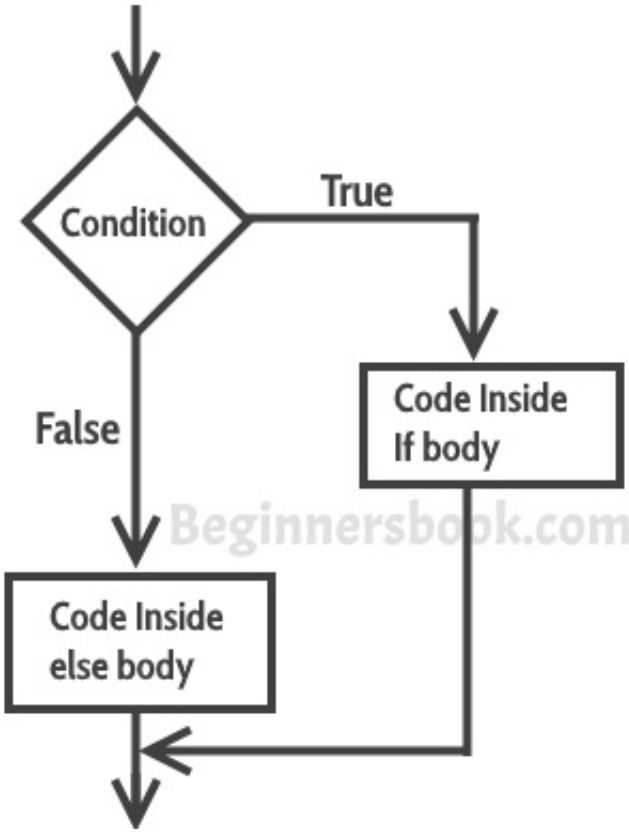
In Javascript, basic flow control structures are:

1. **If** (and **if/else**): they take a **condition** and decide to take one of two blocks of

code.

2. **For** loops: they **repeat** a block of code a number of times.
3. **Switch** statements: like **IF**, but with many paths
4. **While** and **do/while** loops: Similar to **for** loops, they repeat a block of code until a certain condition is reached.
5. **try/catch** blocks: if an error happens, they recover from that error and run a separate block of code

If (conditional blocks)




```
// Basic example
console.log("start!"); // before the condition (always runs)
if (someBooleanValue) { //<- in parenthesis goes the condition, it
can be true or false
  console.log("some boolean value is true!"); // runs only if the
condition is true
}
console.log("start!"); // after the condition (always runs)
```

```
// Example with if/else
console.log("start!"); // before the condition (always runs)
if (someBooleanValue) { //<- in parenthesis goes the condition, it can be true or false
  console.log("some boolean value is true!"); // runs only if the condition is true
} else {
  console.log("some boolean value is false!"); // runs only if the condition is false
}
console.log("start!"); // after the condition (always runs)
```

For loops

If you want to repeat a certain action many times, and you want to have a counter of your repeat, use a **for** loop.

```
// Example: I want to print the numbers 0 to 4;
for (let counter = 0; counter < 5; counter++) {
  console.log(counter);
}
// result:
// 0
// 1
// 2
// 3
// 4
```

The syntax of the for loop:

```
// the for defines one loop, with 0 or more cycles
for   // the keyword "for", followed by three expressions inside a parenthesis, split by ";"
  (
     let counter = 0; // initialisation: run once, before the loop begins
     counter < 5; // condition: evaluted once before each cycle. must be true/false
                  // if true, the next cycle will run. if false, the loop will end
     counter++; // increment: run after each cycle (to add 1 to the counter)
  )
{
  console.log('counter is', counter); // the variable "counter" is available inside the block code
}
```

Why is "for" useful?

To run something only certain number of times:

```
// Imagine we have a function called drawFloweInScreen(positionX, positionY);

// I want to draw 50 flowers in random places of the screen
for (let i = 0; i < 50; i++) {
  drawFlowerInScreen(Math.random(), Math.random());
}

// Imagine that the third argument for drawFlowerInScreen is "lineWidth"

// I want to draw 50 flowers in random places of the screen
for (let i = 0; i < 50; i++) {
  drawFlowerInScreen(Math.random(), Math.random(), i);
}
```

```
// depending of other things, the number 50 might vary; let's make a
function

function drawManyFlowers(totalNumberOfFlowers) {
  for (let i = 0; i < totalNumberOfFlowers; i++) {
    drawFlowerInScreen(Math.random(), Math.random(), i);
  }
}
// later...
drawManyFlowers(50) // 50 flowers will be drawn
drawManyFlowers(80) // 80 flowers will be drawn
drawManyFlowers(finalScore) // one flower per each point in the game
```

To iterate in a list or array

```
function saluteFriends(arrayOfFriends) { // arrayOfFriends should be
an array
  const numberOfFriends = arrayOfFriends.length; // <-- the property
length of an array is a number
  for (let counter = 0; counter < numberOfFriends; counter++) {
    console.log('hello', arrayOfFriends[counter]);
  }
}

saluteFriends(['andre', 'gab', 'tigre']);
```

7: Array

> *An **array** is an ordered list of values that you refer to with a name and an **index**.*

```
// Declare an array
['🐱', '🐶', '🐭'];
// Store an array in a variable
const animals = ['🐱', '🐶', '🐭'];

// print the array
```

```
console.log(animals); // (3) ['🐱', '🐶', '🐻']

// get the number of items in the array
const totalAnimals = animals.length; // the "length" property of an
array is the count of items
console.log(totalAnimals); // 3

// get a specific item. This is the "accessig member" operation
//                      |-------- the name of the variable having the
array
//                      |    ---- the numeric "index" (used with [])
//                      ↓    ↓
const secondAnimal = animals[1];
console.log(secondAnimal); // 🐶
const fifthAnimal = animals[4];
console.log(fifthAnimal);  // undefined
```

1. Accessing a member of an array

```
arrayVar[1]; // This gets the second item of the array stored in the
variable `arrayVar`
```

2. Changing a member on an array

```
const animals = ['🐱', '🐶', '🐻'];
console.log(animals); // (3) ['🐱', '🐶', '🐻']

// we can assign a value to a member just like we do to a variable
animals[1] = '🦘';
console.log(animals); // (3) ['🐱', '🦘', '🐻']

// we can add a value to the array
animals[3] = '🐇';
console.log(animals); // (4) ['🐱', '🦘', '🐻', '🐇'];
```

3. Transforming an array

```
// we can do a lot of operations with arrays
animals.push('🐋'); // adds one item at the end
animals.shift(); // removes the first element

const sortedAnimals = animals.sort(sortingFunction); // sorts an
array
const animalsAndPlants = animals.concat(plants); // concatenates
arrays
const firstTwo = animals.slice(0, 2); // takes a "slice" of the array
animals.map(mapFn); // creates a new array transforming each item of
the original array
```

8: Objects

In programming languages, Object-Oriented Programing (OOP) is a paradigm that helps us to organise code and data together. There are two ways to do OOP: Prototype-based (used by Javascript) and Class-based (used by C# and Java, but also allowed by modern JS). There are many concepts around OOP that we will consider advanced: inheritance, composition, polymorphism... but in Javascript, objects are just a composite data type:

> *An **object** is a collection of properties, and a **property** (aka attribute or field) is an association between a name (or **key**) and a **value**.*

There are a few ways to declare an object, but the most simple is with an **object literal**:

```
// let's assign a new Object to the variable myCar
const myCar = { // in this case, {} indicate not a block of code, but
a literal object
  colour: 'blue', // 'colour' is the property key. 'blue' is the
property value
  year: 2018,   // properties can be of different data types
  isOnSale: false, // in this case, string, number and boolean
};
console.log(myCar); // {colour: 'blue', year: 2018, isOnSale: false}
```

Just like with **arrays**, we can **access** parts of this composite object by using the property name (or **key**):

```
// ... let's examine myCar
const carColour = myCar.colour; // the dot (.) is followed by the key and gives the value
console.log(carColour); // 'blue'

const carColour2 = myCar['colour']; // we can also use [] and the key name as a string
console.log(carColour2); // 'blue'

// since everywhere that we can use a value we can use an expression...
const propertyName = 'colour';
const carColour2 = myCar[propertyName]; // we use a variable to decide what property we want
console.log(carColour3); // blue
```

Properties are very much like variables: we can access them but also we can assign new values to them:

```
myCar.colour = 'red'; // colour will change from 'blue' to 'red'
console.log(myCar); // {color: 'red', year: 2018, isOnSale: false}

// we can also use [] to access the properties
myCar['isOnSale'] = true;
console.log(myCar); // {color: 'red', year: 2018, isOnSale: true}

// we can also add new properties
myCar.make = 'Toyota';
console.log(myCar); // {color: 'red', year: 2018, isOnSale: true, make: 'Toyota'}
```

Finally, the properties can be of any data type, which means they can also be functions (known as methods...)

```
function sayHi() {
  console.log('hi!')
}

myCar.salute = sayHi;
console.log(myCar); // {colour: 'blue', year: 2018, isOnSale: false,
salute: ƒ}
console.log(typeof myCar.salute); // 'function'
myCar.salute();     // hi!
```

And properties can be also objects and arrays, allowing them to have a tree-like structure

```
const otherCar = {
  make: 'Toyota',
  model: 'Yaris',
  details: {              // an object inside an object
    doors: 4,
    hasSeatbelts: true,
    doors: {              // and another level, why not?
      front: 2,
      back: 0,
    }
  },
  revisionYears: [        // an array
    2007,
    2014,
    2015,
    2021,
  ],
  sales: [
    { year: 2007, price: 20000},
    { year: 2015, price: 17000},
    { year: 2022, price: 10000},
  ]
}

// how do we access the data inside?
```

```
console.log(otherCar); // {make: 'Toyota', model: 'Yaris', details:
{…}, revisionYears: Array(4)}
// we access it just like any other property
console.log(otherCar.details); // {doors: {…}, hasSeatbelts: true}
// and we do it level by level:
console.log(otherCar.details.doors.front); // 2
// the consecutive dots and property names can be read like a `path`:
I want the variable `otherCar`, then its property `details` (which is
an object), then its property `doors` (another object), then its
property `front`

// we can mix . and [] access
console.log(otherCar.details['doors'].front); // 2

// for an array, it's the same
console.log(otherCar.sales[2].price); // 10000
```

Classes

Another way to initialise and object is using the keyword `new`. This brings us to the concepts of **Classes**.

Class is the generic of an object. The generic object is an **instance**. The instance **inherits** properties and methods of the class, but can override them.

```
// in Javascript, functions can be used as constructor of new
instances
console.log(Date); // ƒ Date() { [native code] }
const today = new Date(); // Date is the class, the generic. today is
the instance
console.log(today); // Sun Sep 26 2021 07:28:13 GMT+0000

const yearOfToday = today.getYear(); // we didn't define the function
`getYear`: it's inherited from the class Date
console.log(yearOfToday); // 2021

// we can still assign and change properties
today.isGoodDay = true;
console.log(today.isGoodDay); // true
```

In Javascript, we can define our own classes

```
// Let's make a generic class. Lot of new concepts

class Animal {                // the keyword `class` defines a new
Class
  // the special function `constructor` is called when `new` is
called, and its purpose
  // is to initialise the new object (the instance)
  constructor(species) {
    // the keyword `this`, in the context of a class method, is a
reference to the Instance
    this.mySpecies = species; // we create a new property
'mySpecies', and we initialise it
                              // with the parameter of the
constructor
    this.isCool = true;       // Another property, initialised with a
static value
  }
  walk() {                    // We define a method: a property that
is a function
    console.log('I am walking!')
  }
```

```
}

const tigrecito = new Animal('cat'); // we use `new` to create an
Instance of a Class
console.log(tigrecito); // Animal {mySpecies: 'cat', isCool: true}

console.log(tigrecito.mySpecies); // 'cat'
tigrecito.walk(); // 'I am walking!'
```

The area of Object-Oriented Programming is vast and full of concrete and abstract concepts to learn. In Javascript, objects are not class-based, but prototype based. More explanations can be found in https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects

Useful tools for objects

```
// Object.keys gets an array with all the properties of an object
const properties = Object.keys(myCar);
console.log(properties); // (4) ['colour', 'year', 'isOnSale',
'salute']

// we can know the type of a specific property
console.log(typeof myCar.year); // number

// if a property doesn't exist, we'll get undefined
console.log(myCar.model);  // undefined

// careful, if we try to access a deep property that doesn't exist,
it will fail
console.log(myCar.engineDetails);       // undefined
console.log(myCar.engineDetails.model); // Uncaught TypeError: Cannot
read properties of undefined (reading 'model')
```

JSON

Finally, consider that the acronym **JSON** means "**J**ava**S**cript **O**bject **N**otation". An object can be transformed into a string (so it can be stored in a text file, or obtained through an HTTP response) and it can be interpreted from a string (this is called "**parsing**"). The two functions that I have for this are `JSON.stringify()` and `JSON.parse()`.

```
// A regular object
const myDetails = { firstName: 'John', lastName: 'Doe'};
console.log(myDetails); // {firstName: 'John', lastName: 'Doe'}
console.log(typeof myDetails); // object

const myDetailsAsString = JSON.stringify(myDetails); //
{"firstName":"John","lastName":"Doe"}
console.log(typeof myDetails); // string
// now I can store myDetailsAsString into a file, or send it as a
message via HTTP

// ... then later someone who will read this text, or receive it via
a message, can interpret it into a normal object
const contentOfFile = readFileAsync('details.json', 'utf-8'); // this
will read the content of the file
console.log(typeof contentOfFile); // string
const details = JSON.parse(contentOfFile);
console.log(typeof details); // object
console.log(details); // { firstName: 'John', lastName: 'Doe'};

// keep in mind that JSON.parse can fail with an Exception if the
string is malformed
const goodObject = JSON.parse('{}');
console.log(goodObject); // {}
const badObject = JSON.parse('{'); // Uncaught SyntaxError:
Unexpected end of JSON input
console.log(badObject); // Because of the Error above, this line will
never be called
```