Loops

- **While loops**

Similar to if statement, except it repeats a given code block instead of just running it once

Example:

```
var count = 1;

while(count <6){
console.log("Count is : + count);
count ++;
}
```

But also can print character

```
var strg = "helllo";
var count = 0;

while(count < str.length) {
console.log(str[count]);
count++;
}
```

Infinite loops occurs when the terminating condition is never true

- **For loops**

```
for (var count=0; count <6; count++){
console.log(count
}
```

Variables

```
let name = "Ricardo";
console.log(name) => Ricardo
```

Also can be thru **var**

```
var name = "Ricardo";
```

Variables can stores **strings**, **numbers** or **boolean expressions.**
Because it´s a variable it can change.

```
var name = "Ricardo";
name= " John".
```

console.log(name) => “John”

String

let firstName = “Ricardo”;
let lastName = “Moreira”;

console.log(firstName + “ ” + lastName) **=>** Ricardo Moreira

But in ES6

console.log (`${firstName} ${lastName}`); **=>**Ricardo Moreira
or
fullName = (`${firstName} ${lastName}`); **=>** (fullName) => Ricardo Moreira

Strings have a **length** property

“hello.world”.length => 11

To access indivual caracheteres use []
“hello”[0]=>h because javascript starts counting as 0.

Methods

- **toTrim()** => cuts the empty space;
- **length** => number of carachteres;
- **toLowerCase** => all small letters;
- **toUpperCase**t => all big letters;

Types & Values

- String
- Number
- Boolean
- Null and Undefined
- Symbol (ES6)

- Object

Only **object** is not a primitive. A primitive is data that is not an **object** and has no **methods** and is also immutable.
Javascript provides a **typeof** operator that can examine and tell you what type it is:

```
var a = “hello”
typeof a; => string
```

Objects

Object is a compound value where you can set properties (named locations) that each hold their own values of any type.It has key value pairs

```
var obj = {

    a: “hello”,
    b: 42,
    c: true

}

obj.a; = obj [“a”] =>” hello”
```

Arrays

Array is an object that hold value (of any type) in numerically indexed positions.

```
var arr = [
“hello world”,
42,
True
];
arr[0]; // “hello world”
arr[1]; // 42
```

The best and most natural approach is to use **arrays** for numerically positioned values and use **objects** for named properties.

book, webdev,

Functions

Other **object** subtype you will found is a **function.** Functions are executed when they are called, known as invoking a function.
Values can be passed into functions and used within the function. Function always return a value.If no return value is specified, the function will return undefined.

A **Function Declaration** defines a named function. To create a function declaration you use the function keyword followed by the name of the function. When using function declarations, the function definition is hoisted, thus allowing the function to be used before it is defined.

```
        function name(parameters) {
        statements
}
```

A **Function Expressions** defines a named or anonymous function. An anonymous function is a function that has no name. Function Expressions are not hoisted, and therefore cannot be used before they are defined. In the example below, we are setting the anonymous function object equal to a variable.

```
let name = function (parameters){
statements
}
```

An **Arrow Function Expression** is a shorter syntax for writing function expressions. Arrow functions do not create their own this value.

```
let name =  (parameters) =>{
statements
}
```

Parameters are used when defining a function, they are the *names* created in the function definition. In fact, during a function definition, we can pass in up to 255 parameters! Parameters are separated by commas in the (). Here’s an example with two parameters — param1 & param2

```
        const param1= true;
        const param2 = false;

function twoParams(param1, param2){
```

console.log(param1,param2)
}
Arguments, on the other hand, are the *values* the function receives from each parameter when the function is executed (invoked). In the above example, our two arguments are true & false.

- **Sintaxe**
  function calculateAge(yearofBirth) {
  var age = 2018 - yearofBirth;
  return age;
  }

Inside the () is the arguments, the data you want to pass into the function.Is the information we pass thru the function when we run the function.

```
var ageJohn = calculateAge(1990);
console.log(ageJohn);
```

```
function yearsUntilRetirment(name,year){
 var age = calculateAge(year);
var retirement = 65- age;

if (retirement >=0) {
console.log(retirement);
} else {

console.log("You are already retired");
}
}
yearsUntilRetirment('age1', 1990);
```

A **function** inside a **function**

```
ricardo.bar="hello world";
typeof ricardo => function
typeof ricardo() =>"number"
typeof ricardo.bar => "string"
```

Functions are a subtype of **objects.**

➔ **Function Scopes**

The **var** keyword is to declare a variable that will belong to the function scope, if is inside the function. WIll belong to the global scope if is outside the function. That means that a function can access a var outside the function but when out the function you **can´**t access data that is inside the function.

```
var a=2;
foo(); =>3
function foo(){
  a =3;
  //console.log(a);
  var a;
  console.log(a);
}
console.log(a); =>2
foo();=>3
```

Tips

console.table => makes a table out of the data

ES6

Var declaration

const **let** and var

const Can´t be reassigned. By default should use **const**.

const person = "ricardo"
person = "Manuel" // error

let person = "ricardo"
person = "Manuel";
***console*.log** person); // Manuel

Var variables are functions scoped meaning that when a variable is created in a ***function***, everything in that ***function*** can access that variable.

```
function myFunction() {
  var myVar = "Ricardo";
  console.log(myVar;) // Ricardo
}
```

Coercion

Coercion comes in two forms in Javascript : explicit and implicit. Explicit you can see obviously from the code that a conversion from one type to another will occur.
With the implicit the coercion can happen as more of non-obvious.

Explicit coercion:

```
var a = “42”;
var b = Number (a);

a; // “42”
b; // 42 => the number
```

Implicit coercion:

```
var a = “42”;
var b = a *1; // “42” implicitly coerced to 42 here

a; // “42”
b;//42 => the number

var x = 99;
```

x == “99” => **true** because it makes **type coercion**. Changes the number 99 for the string “99”
x === “99” => **false** because === don´t male **type coercion.** checks for number and type and one is a number the other is a string.

```
var y = null;

y == undefined => true;
y === undefined => false
```

So **==** checks for value equality with coercion allowed **===** checks for value equality without coercion allowed. It's called a strict equality.

You should take special note of the **==** and the **===** comparison rules if you're comparing two non-primitive values like **object** (including **function** and **array**).Because these values are held by reference. They check if the references match, not about the values.

For example, arrays are coerced to string by simply joining all the values by commas. So it would seem that two arrays with the same contents are the same but are not.

```
var a = [1,2,3];
var b= [1,2,3];
var c= “1,2,3”;

a==c =>true;
b==c =>true;
a ==b => false;
```

Operator Precedence

```
var ageJohn = 30;
var ageZen = 30;

ageJohn = ageZen = (3+5) * 4 - 6;


console.log(ageJohn); //26
console.log(ageZen); //26

ageJohn++;
//ageZen = ageJohn;
console.log(ageJohn); //27
console.log(ageZen); //26

ageJohn = ageZen;

console.log(ageJohn); //26
console.log(ageZen); //26
```

This happens because the operator precedence. For example for assignment it goes **right to left.**

Truthy & Falsy

true == “1” => **true**
true ==”2” => **false**
0==false => **true**
0== “” => **true**
null ==undefined => **true**
NaN == Nan => **false**
“” == false **=> true**

Falsy Values:

- false
- 0
- “”
- null
- undefined
- NaN

A non-boolean value only follows the truthy & falsy coercion if it's actually coerced to a boolean. **Null** and **undefined** are only equal to themselves.**NaN** is not equivalent to nothing, not even itself

Methods

Alert
Pops up a message to the user

alert(“Hello There”);

Prompt

Asks for a command or info, accordingly with what we ask for.

prompt(“What is your name”)

console.log

Prints to the Javascript console

console.log(“hello there”) => Prints only to the console, not to the user

If and Else Statement

```
var name = "Ricardo";
var age = 20;
var isMarried = true;

if (isMarried === false) {
 console.log(name + " " + "is married");
} else {
 console.log(name + " " + "will marry soon");


if (age < 20) {
  console.log(“you are a teenager”);
} else if  (age >=20 && age < 30 {
   console.log(“you are a young man”);
} else {
  console.log( “you are a man”);
}
```

But if there is too many **else if:**

```
var job =”teacher”;

switch(job) {
  case ‘teacher’ {
    console.log(“He teaches kids”);
  break; // in case if it´s true it´s breaks(stops) here, don´t continue.
case ‘driver’ {
console.log(“he drives a taxi”);
```

```
break;
case ‘cop {
console.log(“he is not the batman”);
break;
default:
console.log(“he does something else”);
}
```

Arrays

```
var names =[‘John’, ‘Ricardo’, ‘Manel’];
var years = [‘1980’,’2002’,’1997’];
```

source:

js udemy

you dont know js

webdev bootcamp