

Unit 2

BASIC JAVASCRIPT SYNTAX AND STRUCTURES

DESARROLLO WEB EN ENTORNO CLIENTE 24/25

ALEJANDRO VIANA RÍOS



Unit 2

Basic Javascript syntax
and structures



Index

1. Code quality
2. Interacting with user
3. Data types
4. Variables
5. Data type conversions
6. Operators
7. Conditional execution
8. Loops
9. Functions

1.- Code quality

Code quality

1.- Code quality

- JavaScript is really flexible but for the sake of debuggin some tips should be followed
- Some code-style makes easier to understand, modify and debug a program

```
1 //Bad coding style. Confusing for newers. Error-prone
2
3 i=5
4 j=0
5 i != j ? i > 0 ? console.log(Math.max(i, j)) : console.log(i) : console.log(0);
6
7 //Same code than above. Brackets could be removed, but for the sake of clarity, it's better to leave them
8 let i=5;
9 let j=0;
10 if ( i!=j ){
11   if ( i>0 ){
12     console.log( Math.max( i,j ) );
13   }else{
14     console.log( i );
15   }
16 }else{
17   console.log( 0 );
18 }
```

Lines 3-5 do the same than 8-18, but the latter are much easier to understand and debug

Variables, sentences and comments

1.- Code quality

- Variables
 - Use descriptive and different names. Caution with date and data
 - Declare them with let and initialize at creation, if possible.
 - Do not change them type during their life
 - Do not reuse variables. Declare new ones
 - Use const when a variable is not changing its value ever
- Sentences
 - Do not split one sentence into several lines
 - Place one sentence per line. Do not place several sentences in the same line (although using semicolons)
 - Use semicolons to finish each sentence although JavaScript automatically adds it
- Comments: A good code should be almost self-explanatory, huge comments explaining everything are a symptom of bad coding

Functions

1.- Code quality

- If, by reading its name, it is supposed not to have any side-effect, it should not! (isPresent, checkPermission, etc.)
- Avoid returning non standard values like objects in functions like checkPermission (should return T/F)
- They should do what is advised on their name. Nothing less, nothing more. One function, one action
- Try to replace code with functions

```
1 function showPrimes(n) {  
2   nextPrime:  
3   for (let i = 2; i < n; i++) {  
4     // check if i is a prime number  
5     for (let j = 2; j < i; j++) {  
6       if (i % j == 0) continue nextPrime;  
7     }  
8     alert(i);  
9   }  
10 }  
11 }  
12 }
```

```
1 function showPrimes(n) {  
2  
3   for (let i = 2; i < n; i++) {  
4     if (!isPrime(i)) continue;  
5     alert(i);  
6   }  
7 }  
8  
9  
10 function isPrime(n) {  
11   for (let i = 2; i < n; i++) {  
12     if (n % i == 0) return false;  
13   }  
14  
15   return true;  
16 }
```

Replacing code with function makes it easier to understand and maintain

Functions

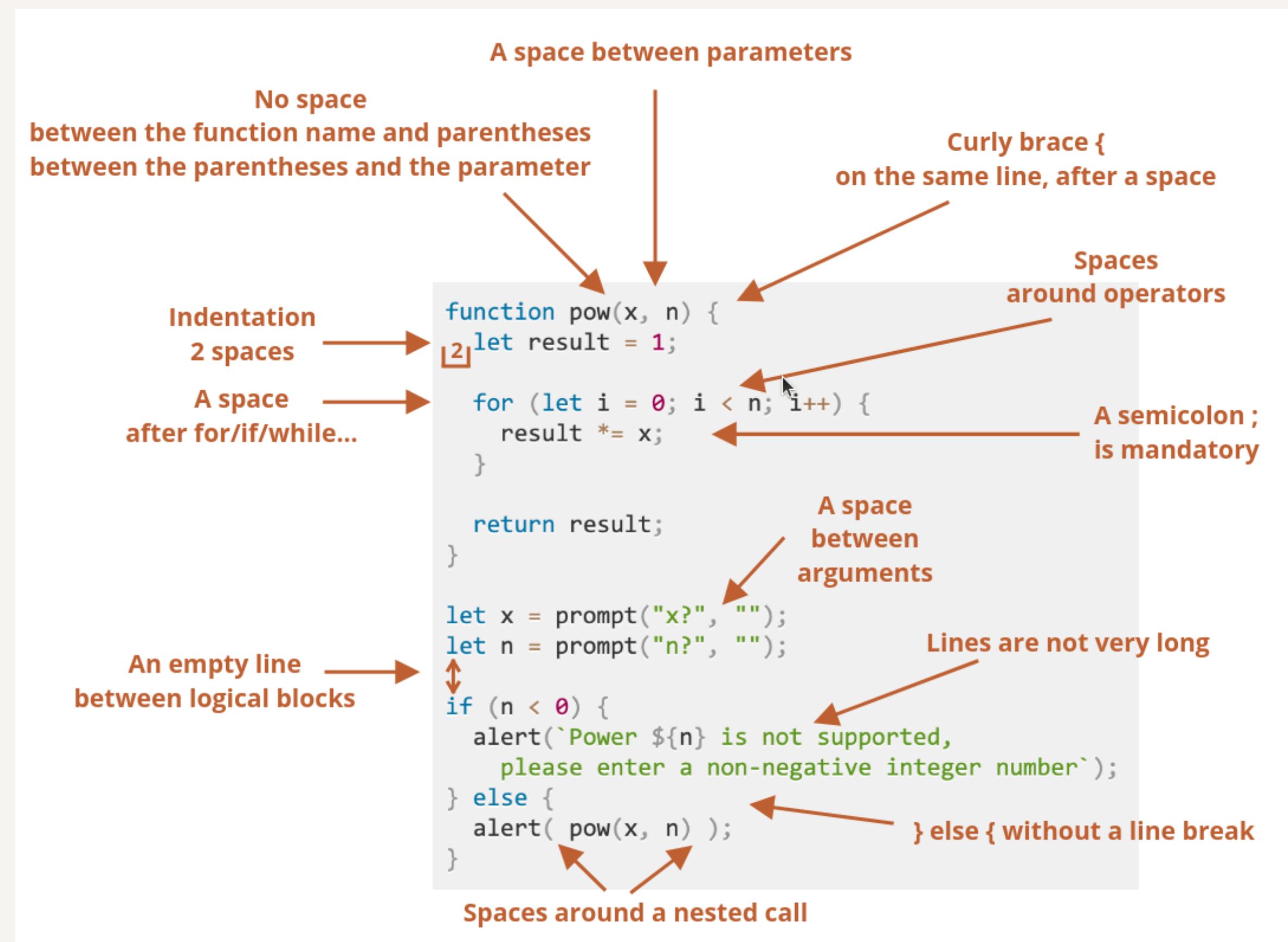
1.- Code quality

- Try to abstract code to avoid creating similar ones like, for instance, printScreen and printPrinter
- Describe what they do, their inputs and outputs. Some tools can automatically generate documentation (like jsdoc)
- Describe why the problem was solved this way and if there are more ways for solving it

```
1  /**
2   * Returns x raised to the n-th power.
3   *
4   * @param {number} x The number to raise.
5   * @param {number} n The power, must be a natural number.
6   * @return {number} x raised to the n-th power.
7   */
8  function pow(x, n) {
9    ...
10 }
```

Code quality

1.- Code quality



Recommended coding way

Notation

1.- Code quality

- Keep consistency when using notation
 - Camel case: contarPalabrasImpares
 - Pascal case: ContarPalabrasImpares
 - Snake case: contar_palabras_impares
 - Kebab case: contar-palabras-impares

2.- Interacting with users

Interacting with user

2.- Interacting with users

script.js ×

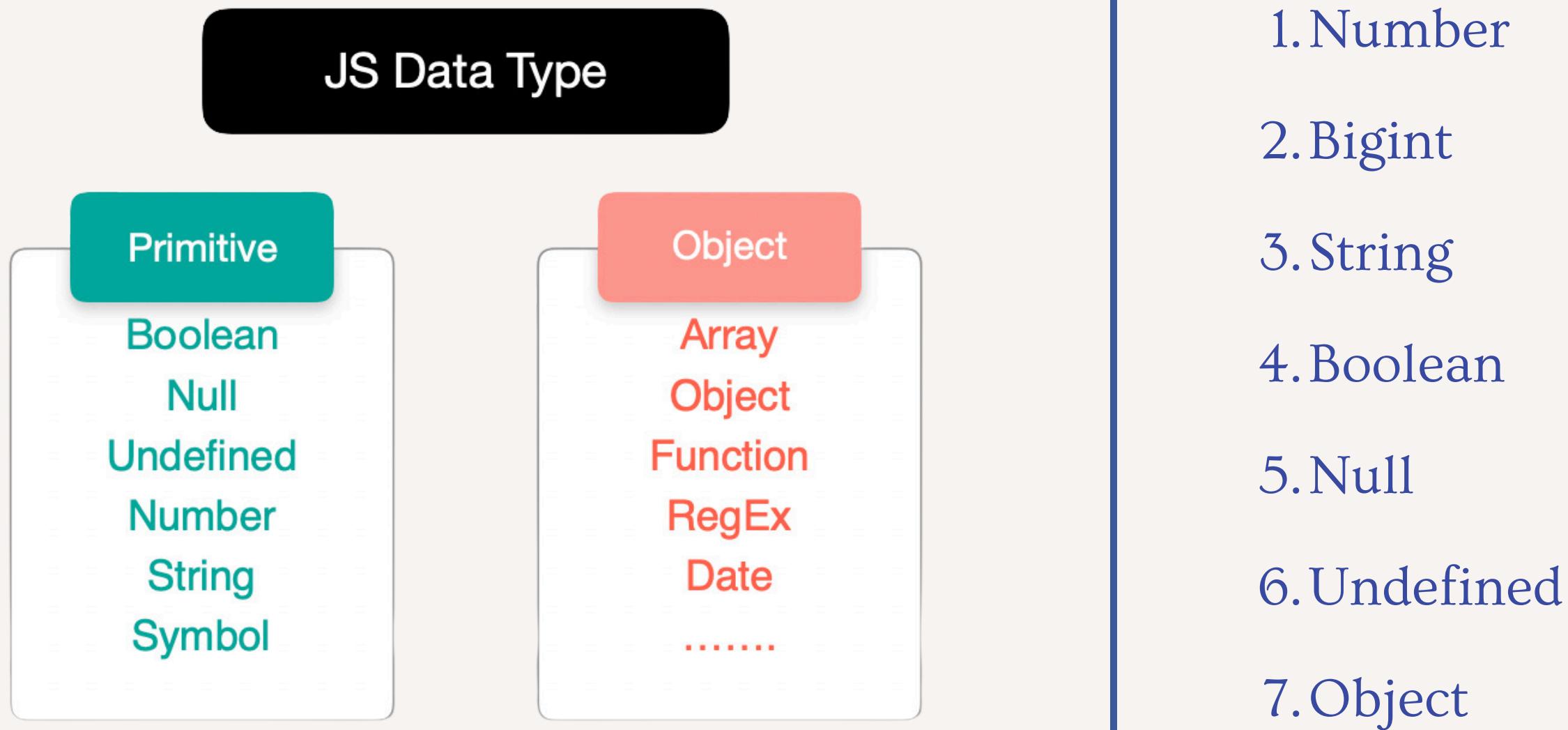
```
1 //shows up a modal window with the message and waits for the user to press ok
2 alert ("esto es un mensaje");
3
4 //shows up a modal window asking the user for a value
5 let numero=prompt("dame un número", "numero");
6
7 //shows a message at the console
8 console.log("esto es un texto");
9
10 //shows up a modal window asking the user for confirmation
11 let respuesta=confirm("¿acepta?");
```

Console ×

```
estos es un texto
```

4 ways of interacting with users: alert, console, confirm and prompt

3.- Data types



Primitive data types

3.- Data types

Number

- Integer or floating point
- Special values: Infinity, -Infinity, NaN (not a number)
- erroneous operations can be done and JavaScript will never stop (you will get NaN)

BigInt

- Allows to represent integers larger than $(2^{53})-1$
- Is created by appending a “n” at the end of an integer number

String

- Chain of symbols (letters, numer, special symbols), treated as one
- Must be surrounded by double, simple quotes or backticks
- Double or simple quotes are the same. Backticks are extended functionality, allowing to embed variables or expresions by wrapping them in \${var}

Primitive data types

3.- Data types

Boolean

- Logic value that can be true or false

Null

- Intentional lack of value
- Developer assigns to a recently created variable

Undefined

- It is also a lack of value, but unintentionally
- Developer just creates variable and assigns no value

Non-primitive data types: Objects

3.- Data types

- They are a category of complex data types
- They can contain data collections (pair key-value) and functionality (methods)
- There are some kinds of objects:
 - simple ones as {}
 - complex ones as arrays, classes or functions

Objects

3.- Data types > Non-primitive data types

Array

- It's a subtype of an object
- Ordered list of values of any type enclosed into brackets
- [1, 2, 3, "pepe", "jose", true]

Function

- An object representing an executable chunk of code that can be called anywhere
- `function sumar (a,b){ return a+b;}`

Date

- An object to work with dates and time
- `new Date();`

RegExp

- An object to work with regular expressions
- `/\d+/-`

Iterables and non-iterables

3.- Data types

Iterables

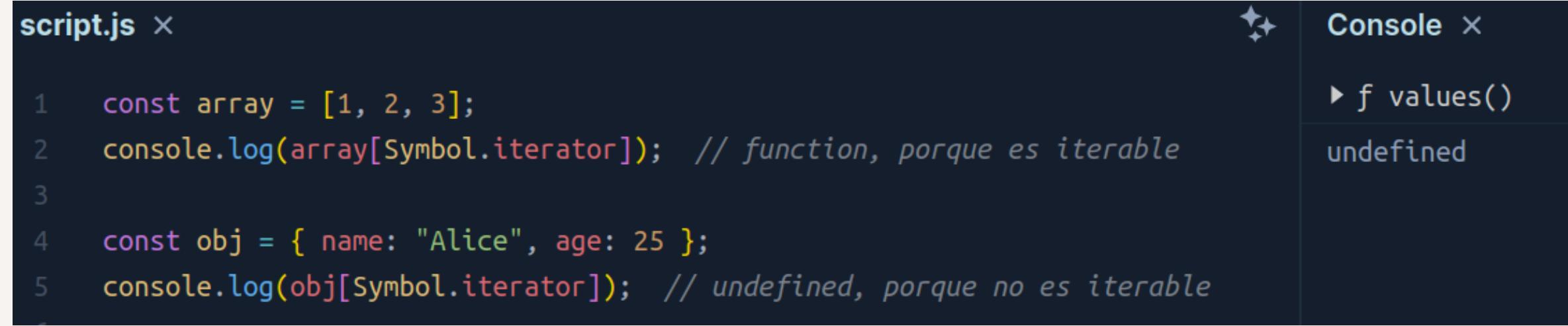
- Their elements can be sequentially accessed by using iterative structures like for...of, spread operators (...) or methods like Array.from()
- Implements Symbol.iterator method
- Examples: Arrays, Strings, Maps, Sets, NodeLists
(document.querySelectorAll())

Non iterables

- Their elements can be accessed by using for...in or Object.keys()
- Don't implement Symbol.iterator method
- Examples: Objects, numbers, booleans or Errors

Iterables

3.- Data types



The screenshot shows a code editor window titled "script.js" and a developer tools console window titled "Console". The code in "script.js" is as follows:

```
script.js ×

1 const array = [1, 2, 3];
2 console.log(array[Symbol.iterator]); // function, porque es iterable
3
4 const obj = { name: "Alice", age: 25 };
5 console.log(obj[Symbol.iterator]); // undefined, porque no es iterable
```

The "Console" window shows the output of the log statements:

```
Console ×
▶ f values()
undefined
```

A caption below the code editor reads: "Iterable objects implements Symbol.iterator method".

Iterable objects implements Symbol.iterator method

4.- Variables

Recommendations

4.- Variables

- JavaScript is a weakly typed language: User does not need to assign any type when creating the variable and it can change (not recommended)
- It is recommended not to change variable type
- It is recommended to initialize a variable when creating
- It is only allowed to use number, letters, _ and \$ for naming

Creation

4.- Variables

```
let numero1=5, numero2=10, cadena="hola don pepito";
let numero3=1,
    numero4=2,
    numero5=3;
let numero6=4;
let numero7=5;
```

Creating variables

Creation

4.- Variables

```
1 let aux="hola";
2 let aux2=aux;    //aux value is copied into aux2 value. Both point to a different memory location
3
4 aux2="adios";   //if I modify aux2, aux still holds its original value
5 console.log(aux, aux2);
```

Console ×

```
hola adios
```

When duplicating variables, they point to different memory locations. They are different variables. It won't happen the same with objects

Variables

4.- Variables

- Use human-readable names like `userName` or `shoppingCart`.
- Stay away from abbreviations or short names like `a`, `b`, `c`, unless you really know what you're doing.
- Make names maximally descriptive and concise. Examples of bad names are `data` and `value`.
- Be consistent. Should a site visitor is called a “user”, related variables should be named `currentUser` instead of `currentVisitor`
- Always use lowercase except with constants whose value is known prior to execution

Creation

4.- Variables

```
1 let numero1=5;
2 let numero2=10;
3 let cadena="hola don pepito";
4 let asignaturas=["Desarrollo web en entorno cliente", "Diseño de interfaces web"];
5 let persona= {nombre:"procopio", apellido: "Maximus"};
6 let constante="esto no debería cambiar";
7
8 console.log ( numero1 + cadena );
9 console.log ( `La suma de ${numero1} y ${numero2} da ${numero1+numero2}` );
10 console.log ( numero1+numero2 );
11 console.log( asignaturas[1] );
12 console.log( persona.nombre )
13 console.log ( typeof(numero1), typeof(cadena), typeof(asignaturas), typeof(persona), typeof( constante ) );
14
```

Console ×

5hola don pepito

La suma de 5 y 10 da 15

15

Diseño de interfaces web

procopio

Strings declaration

4.- Variables > Strings

The screenshot shows a code editor with two tabs: 'script.js' and 'Console'. The 'script.js' tab contains the following code:

```
script.js ✘
1 let single = 'single-quoted';
2 let double = "double-quoted";
3 let backticks = `backticks`;
4
5 let suma=1+2;
6 console.log (`suma vale ${suma}`);
7
8 let lista= `Lista de compra:
9     *Pepinillos
10    *Tomates
11    *Calabacines`; //allowed
12
13 /*
14 let lista2= "Lista de compra:
15     *Pepinillos
16     *Tomates
17     *Calabacines"; //not allowed*/
```

The 'Console' tab shows the output of the code:

```
Console ✘
suma vale 3
```

Some ways of declaring strings

5.- Data type conversion

String conversion

5.- Data type conversions

script.js ×

```
1 //Mal por llamar cadena a un booleano
2 let cadena = true;
3 console.log (typeof cadena); // boolean
4
5 cadena = String(cadena) // now value is a string "true"
6 console.log (typeof cadena); // string
```

Console ×

```
boolean
string
```

Typeof checks variable data type and String converts to a String

Number conversion

5.- Data type conversions

```
script.js ×  
1  console.log ( "4"/"2" );  
2  let cadena2="123";  
3  
4  console.log (typeof(cadena2));  
5  let num=Number(cadena2);  
6  console.log (typeof(num));
```

```
Console ×  
2  
string  
number
```

Integer conversion

Boolean conversion

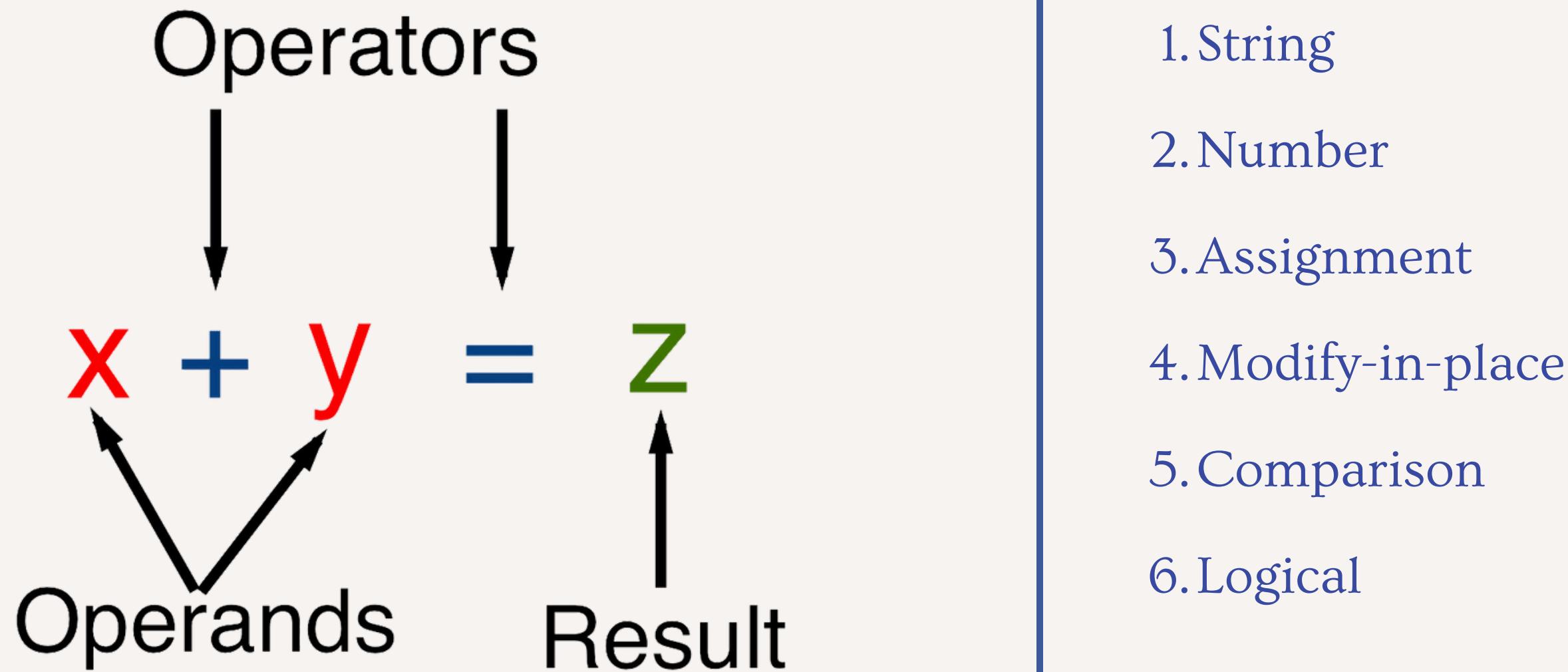
5.- Data type conversions

The screenshot shows a browser's developer tools interface with two panes: 'script.js' on the left and 'Console' on the right. The 'script.js' pane contains a script with numbered comments explaining Boolean conversion rules. The 'Console' pane shows the output of each log statement.

Line	Code	Output
1	//Boolean conversion	
2	/*Values that are intuitively “empty”, like 0, an empty string, null, undefined, and NaN, become false.*/	
4	console.log (Boolean(NaN));	false
5	console.log (Boolean(undefined));	false
6	console.log (Boolean(0));	false
7	console.log (Boolean("")); //any number distinct than 0 becomes true	true
9	console.log (Boolean(11123123));	true
10	console.log (Boolean(1));	true

Boolean conversion

6.- Operators



String operators

6.- Operators

```
script.js ×

1 //string contatenation
2 let cadena="hola",
3     cadena2="amados",
4     cadena3="alumnos",
5     cadena4='7',
6     cadena5='10';
7 console.log(cadena+" "+cadena2+" "+cadena3);
8 console.log(5+5+'4');

9
```

Console ×

```
hola amados alumnos
104
```

String concatenation

Number operators

6.- Operators

The screenshot shows a code editor with two tabs: 'script.js' and 'Console'. The 'script.js' tab contains a block of JavaScript code with numbered lines from 1 to 19. Lines 1 through 15 are standard arithmetic operations. Line 16 is a comment about operator precedence. Lines 17 through 19 demonstrate different ways to calculate the expression $(3+10) \cdot 2$. The 'Console' tab shows the output of each line, with the last three lines showing the results of the commented-out calculations.

```
script.js ×

1 //Math operators: +, -, *, /, % (remainder, resto), ** (exponentiation, exponente)
2 let cadena4='7',
3     cadena5='10';
4 let numero=5,
5     numero2=12;
6 console.log (numero2%numero);
7 console.log (numero**2);
8 console.log(+cadena4 + -cadena5);    //cadenas are converted into numbers thanks to the +/- operator
9 let numero_negativo=-numero;
10 console.log (numero_negativo);

11
12 console.log(numero2);
13 numero2-=numero;
14 console.log(numero2);

15
16 //Be careful with operators precedence. It may change the result of the operation
17 let x=2*2+1;
18 console.log(x);
19 console.log(3+10*2, 3+(10*2), (3+10)*2);

Console ×

2
25
-3
-5
12
7
5
23 23 26
```

Assignment operator

6.- Operators

script.js ×

```
2 //Assignment
3 //= is also an operator, returning a value. Expressions are evaluated from right to left, taking into account operator precedence
4 let a=4;
5 let b=5;
6 let c=4-(a=b+1);    //confusing syntax. Not recommended
7 let d=a=b=4+3;
8 console.log (c, d);
```

Console ×

```
-2 7
```

Assignment operator

Modify-in-place operators

6.- Operators

The screenshot shows a browser developer tools interface with two panels: 'script.js' on the left and 'Console' on the right.

script.js content:

```
1 let n=5;
2 let c;
3 c=n--; //first n value is returned and then n value is decremented
4 console.log(c);
5 c=--n; //first n value is decremented and then n value is returned
6 console.log(c);
7
8 console.log(n=n+5, n+=5);
9 console.log(n);
10 console.log(n*=2);
11 console.log(n++, ++n);
12 console.log(n);
13 console.log(n--, --n);
```

Console output:

Line	Value
1	5
2	3
3	8 13
4	13
5	26
6	26 28
7	28
8	28 26

Modify-in-place operator

Comparison

6.- Operators

The screenshot shows a code editor with a dark theme. On the left, a file named `script.js` is open, containing 36 lines of JavaScript code. On the right, a "Console" window displays the output of the code.

script.js Content:

```
script.js ×

1 let num1=3,
2   num2=num3=5;
3
4 let cad1="hola",
5   cad2="Hola",
6   cad3="holas";
7
8 //unicode is used to determine the order of the letters
9 console.log( num1>num2 );
10 console.log( num2~~=num3 );
11 console.log( num1~~!=num2 );
12 console.log("-----");
13 console.log( cad1>cad2 );
14 console.log( cad1~~==cad2 );
15 console.log( cad1==cad2 );
16 console.log("-----");
17 console.log( "4">num1 );
18
19 /*Equality cannot differentiate 0 or empty string from false
20 due to the fact that empty string, and false, are converted to zero*/
21 console.log("-----");
22 console.log( 0~~==false );
23 console.log( ""~~==false );
24
25 /*therefore, if differentiate is needed, strict equality must be used
26 strict equality do not convert any data type so, if they are not the same
27 type, it returns false*/
28 console.log("-----");
29 console.log( 0===false );
30 console.log( ""===false );
31
32 //be carefull comparing with a variable that could take null or undefined value
33 console.log("-----");
34 console.log( a>4 );
35 let a=3;
36 console.log( a>4 );
```

Console Output:

```
false
true
true
-----
true
false
Hola
-----
true
-----
true
true
-----
false
false
-----
false
false
```

Logical operators

6.- Operators

The screenshot shows a code editor with two tabs: 'script.js' and 'Console'. The 'script.js' tab contains a script demonstrating logical operators. The 'Console' tab shows the output of running the script.

```
script.js ×

1  //=====logical operators=====
2  let a = (d = true),
3      b = (c = false); //remember that 0, empty string, Null or NaN are false, 1 or higher are true
4
5  //or operator //
6  console.log(a || b, b || c);
7  console.log(a || b || c); //should any operand be true, evaluation is stopped and true is returned
8  if (a || b) {
9      console.log('alguno de los dos es verdadero');
10 } else {
11     console.log('los dos son falsos');
12 }
13 console.log('-----');
14
15 //or processes values until first true valor is found, returning the argument and leaving the unexplored
16 console.log(a || b || c || 'ninguno');
17 c || console.log('texto'); //should c be false, texto would show up
18 console.log('-----');
19
20 //and operator (&&)
21 console.log(a && b, b && c);
22 console.log(a && b && c); //should any operand be false, evaluation is stopped and false is returned
23 a && console.log('texto'); //if a exists, texto is shown
24 a > 0 && console.log('a es mayor que cero'); //if a is greater than zero, a text is shwon. No recomendable: poco legible
25 console.log('-----');
26
27 //not operator (!)
28 console.log(!(c || d) && (a || b));
```

Console ×

Output
true false
true
alguno de los dos es verdadero

true
texto

false false
false
texto
a es mayor que cero

false

Logical operators

6.- Operators

The screenshot shows a browser's developer tools interface with two tabs: 'script.js' and 'Console'. The 'script.js' tab contains a block of JavaScript code with comments explaining the nulish operator. The 'Console' tab shows the output of the code, which includes the value '100' and the value '0'.

```
script.js ×

1 //=====nulish operator=====/
2 // || (or) do not distinguish, 0, empty string "", and null/undefined, they are all "false" values
3 //?? (nulish) returns the first argument that is not null nor undefined
4 // when using with ?? or ||, parenthesis must be used
5 let height = 0; // altura cero
6
7 //useful to assing default values to variables
8 console.log(height || 100);
9 console.log(height ?? 100);
```

script.js	Console
//=====nulish operator=====/ // (or) do not distinguish, 0, empty string "", and null/undefined, they are all "false" values //?? (nulish) returns the first argument that is not null nor undefined // when using with ?? or , parenthesis must be used let height = 0; // altura cero //useful to assing default values to variables console.log(height 100); console.log(height ?? 100);	100 0

Nulish coalescing (coalescencia nula) allows to distinguish between 0 and null

7.- Conditional execution

Condition is true

```
let number = 2;  
if (number > 0) {  
    // code  
}  
else {  
    // code  
}  
// code after if
```

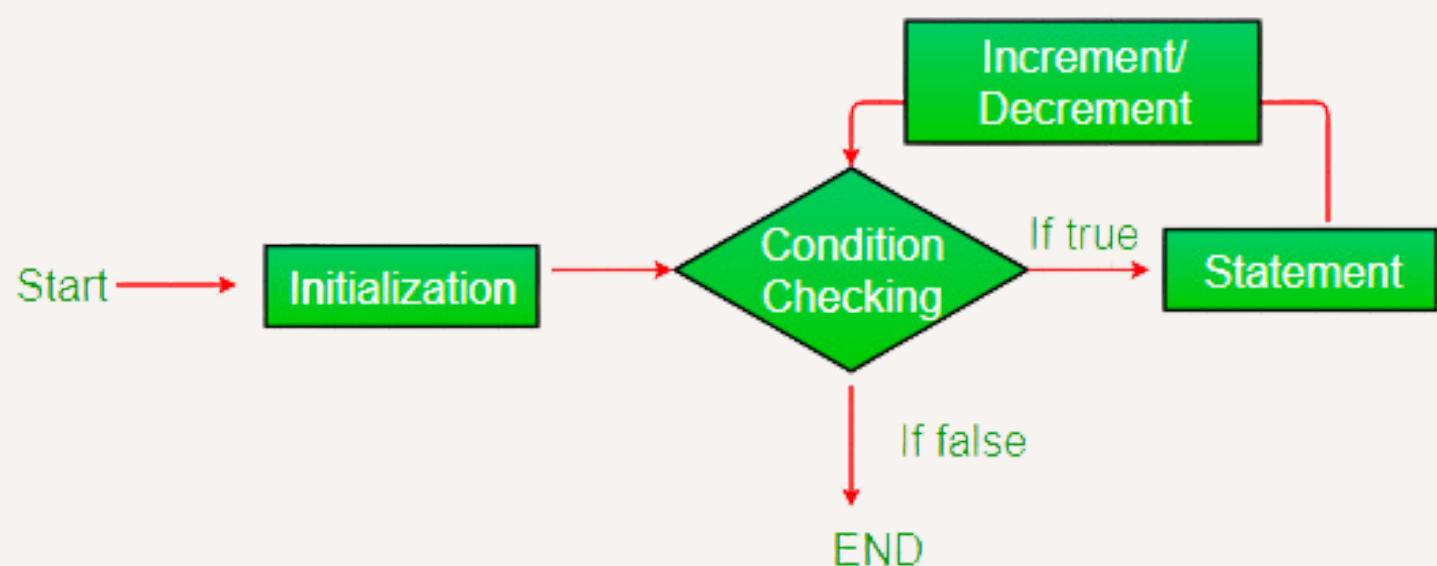
Condition is false

```
let number = -2;  
if (number > 0) {  
    // code  
}  
else {  
    // code  
}  
// code after if
```

1.if..else

2.Ternary operator

3.switch



if...else

7.- Conditional execution

The screenshot shows a code editor with two tabs: "script.js" and "Console". The "script.js" tab contains the following JavaScript code:

```
script.js ×

1  let anyos=prompt("¿Cuántos años tienes?", "año");
2  let mayoria_edad= (anyos>=18);
3  let fechoria_cometida=true;
4
5  if (mayoria_edad){
6      console.log("Eres mayor de edad");
7  }else{
8      console.log("eres menor de edad");
9  }
10
11 //if
12 if (mayoria_edad && fechoria_cometida){
13     console.log("A la cárcel directo");
14 }else{
15     if (fechoria_cometida){
16         console.log ("Al reformatorio");
17     }else{
18         console.log("Sigue así, respeta la ley");
19     }
20 }
```

The "Console" tab shows the output of the script:

```
Console ×

Eres mayor de edad
A la cárcel directo
```

Conditional branching

Variable scope

7.- Conditional execution

The screenshot shows a browser's developer tools. At the top, there is a script editor window titled "script.js" containing the following code:

```
script.js ×

1  if (1){
2    let cadena="hola";
3    console.log(cadena); //prints hola
4  }
5  console.log (cadena); //error! cadena is not defined
6
```

Below the script editor is a "Console" window. It has two entries:

- "hola" (in light blue, indicating it was typed by the user)
- A red error message:
 - ▶ ReferenceError: cadena is not defined
 - @<https://preview-javascript.playcode.io/> line 19 > injectedScript:64:13
 - mn@<https://preview-javascript.playcode.io/> line 6 > injectedScript:19:5455

cadena is a local variable and only exists within the if

Ternary operator

7.- Conditional execution

The screenshot shows a code editor with two tabs: 'script.js' and 'Console'. The 'script.js' tab contains the following code:

```
script.js ×

1  let anyos=prompt("¿Cuántos años tienes?", "año");
2  let mayoria_edad= (anyos>=18);
3  let fechoria_cometida=true;
4
5  if (mayoria_edad){
6      console.log("Eres mayor de edad");
7  }else{
8      console.log("eres menor de edad");
9  }
10
11 //if
12 if (mayoria_edad && fechoria_cometida){
13     console.log("A la cárcel directo");
14 }else{
15     if (fechoria_cometida){
16         console.log ("Al reformatorio");
17     }else{
18         console.log("Sigue así, respeta la ley");
19     }
20 }
21
22 /*? operator works as an if. It is shorter and appealing, but less readable.
23 Be carefull, it's not very intuitive when nesting*/
24
25 //Equivalent to first if
26 ( mayoria_edad ) ? console.log("Eres mayor de edad") : console.log("Eres menor de edad");
27
28 //Equivalent to second if
29 (mayoria_edad && fechoria_cometida) ? console.log("A la cárcel directo") :
30     fechoria_cometida ? console.log("Al reformatorio") :
31         console.log("Sigue así, respeta la ley");
```

The 'Console' tab shows the output of the code:

```
Eres mayor de edad
A la cárcel directo
Eres mayor de edad
A la cárcel directo
```

Conditional branching

switch

7.- Conditional execution

The screenshot shows a code editor with two tabs: 'script.js' and 'Console'. The 'script.js' tab contains the following code:

```
script.js ×

1 //switch
2 //this is the only structure that needs break statement
3 let nota = Number(prompt("¿Qué nota sacaste?",0));
4 switch (nota) {
5     case 0:
6     case 1:
7     case 2:
8     case 3:
9     case 4:
10    console.log( 'Tienes que estudiar más' );
11    break;
12    case 5:
13    console.log( 'Por los pelos' );
14    break;
15    case 6:
16    case 7:
17    case 8:
18    console.log( 'Bien' );
19    break;
20    default:
21    console.log( "¡Excelente!" );
22 }
```

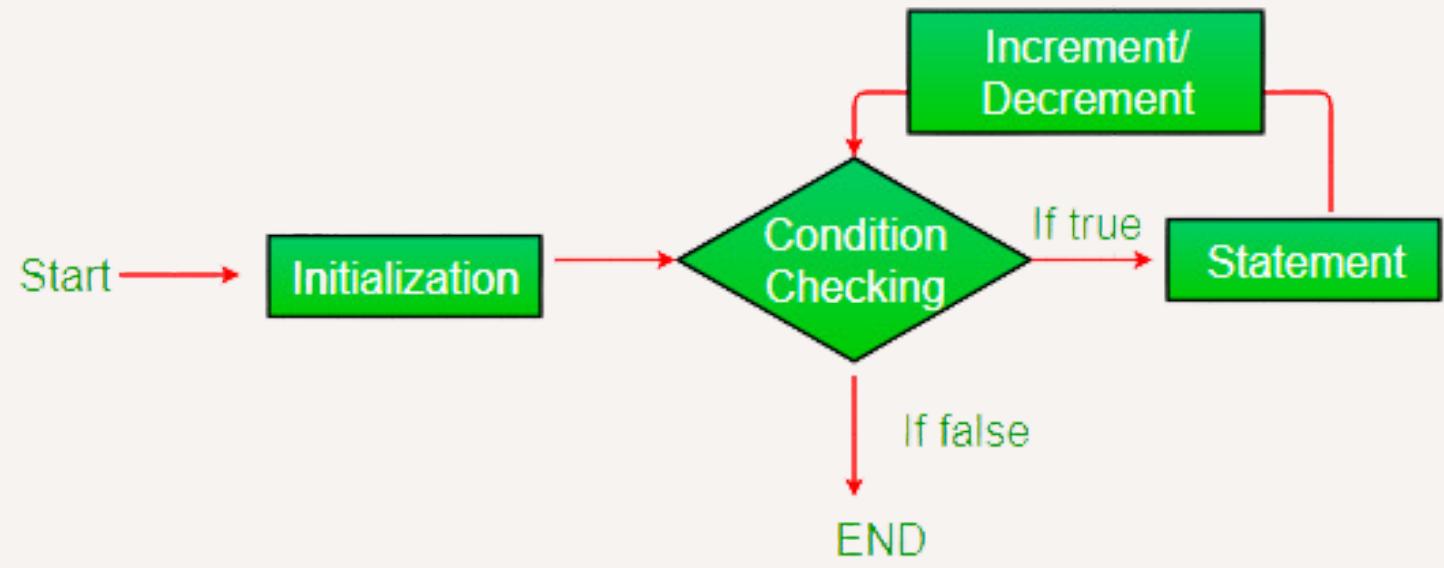
The 'Console' tab shows the output of the script:

```
Console ×

Tienes que estudiar más
```

switch statement

8.- Loops



- 1.while
- 2.for
- 3.Continue
- 4.Break
- 5.Break labels

do...while

8.- Loops

```
//do...while
//Code inside the loop is executed, at least, 1 time before checking condition
let l = 0;
do {
| console.log( l++ );
} while (l <= 3);

do{
| //something
} while (1); //infinite loop
```

do...while

While

8.- Loops

```
//code inside the loop is executed only if the condition is true
let i = 0;
while (i < 3) {
| console.log( i++ ); //it outputs i value and then increment it
}

let j = 0;
while (j < 3) {
| console.log( ++j ); //it increments j value and then shows it
}

let k = 5;
while (k) { //elegant way to write k>0
| console.log( k-- );
}

while (1){ //infinite loop
| //something
}
```

while

for

8.- Loops

The screenshot shows a code editor with two tabs: 'script.js' and 'Console'. The 'script.js' tab contains the following code:

```
1 //nested loops
2 for (let m = 0; m < 3; m++) { // shows 0, then 1, then 2
3   for (let n = 0; n < 3; n++) { // shows 0, then 1, then 2
4     console.log(m,n);
5   }
6 }
7
8 //endless for
9 for (;){
10   console.log("this loop will never end, unless you press ctrl+c");
11 }
```

The 'Console' tab shows the output of the code:

Line	Output
1	0 0
2	0 1
3	0 2
4	1 0
5	1 1
6	1 2
7	2 0
8	2 1
9	2 2
10	this loop will never end, unless you press ctrl+c
11	this loop will never end, unless you press ctrl+c
12	this loop will never end, unless you press ctrl+c
13	this loop will never end, unless you press ctrl+c
14	this loop will never end, unless you press ctrl+c
15	this loop will never end, unless you press ctrl+c
16	this loop will never end, unless you press ctrl+c

nested and endless loop

for

8.- Loops

script.js ×

```
2 const matriz= [1, 2, 3];
3 const obj = { name: "Alice", age: 25 };
4
5 //an array
6 for (let i=0; i<matriz.length; i++){
7   console.log(matriz[i]);
8
9 //an object. It needs to use Object.keys(objeto), a predefined method. We'll talk about methods of predefined objects in a later chapter
10 //Object.keys returns an array with the keys of the object
11 let numero_llaves=Object.keys(obj).length
12 for (let i=0; i<numero_llaves; i++){
13   let llave=Object.keys(obj)[i];
```

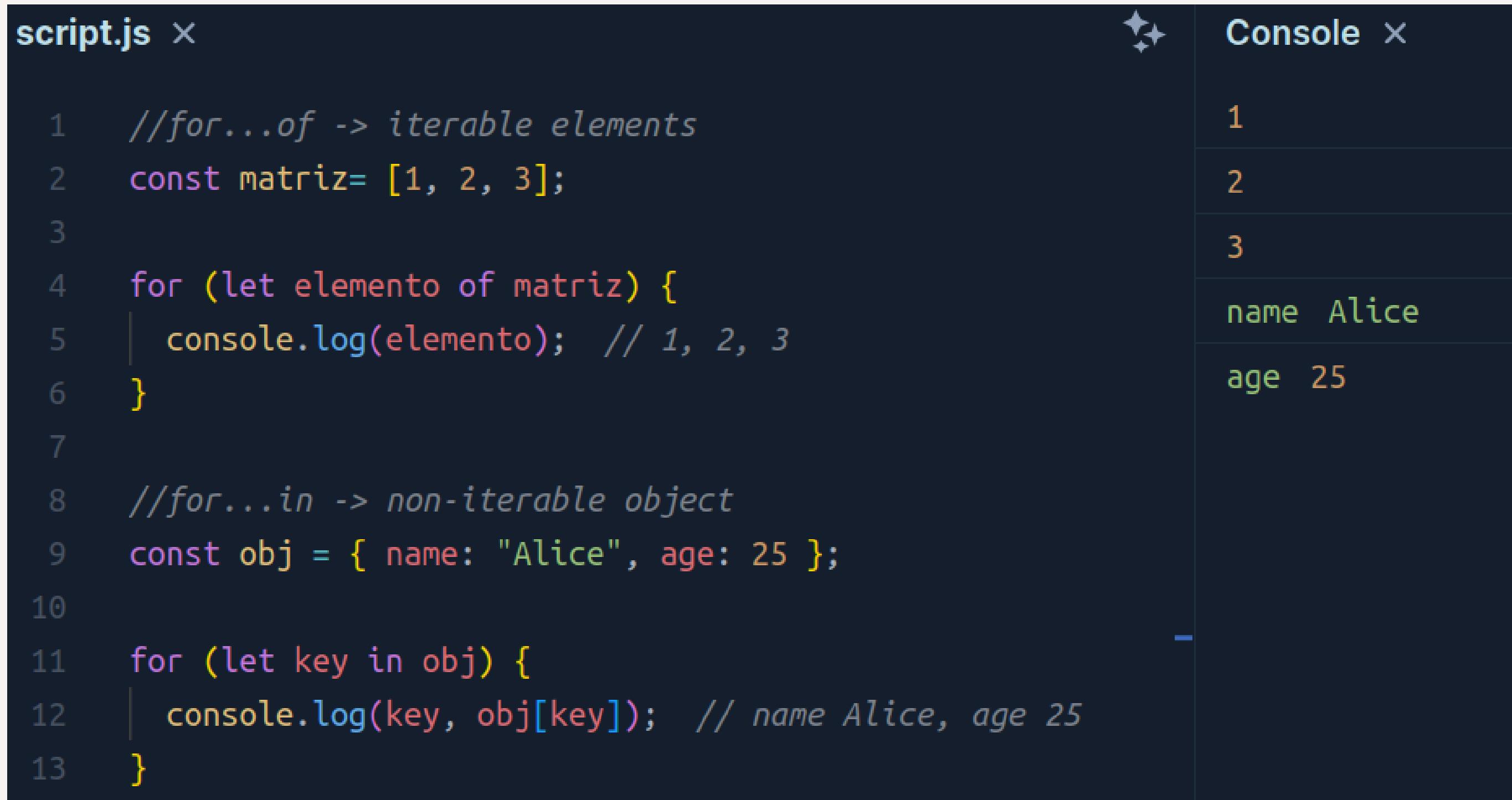
Console ×

```
1
2
3
Alice
25
```

iterating through objects using regular for

for

8.- Loops



The screenshot shows a code editor interface with two tabs: "script.js" and "Console".

script.js tab:

```
script.js ×

1 //for...of -> iterable elements
2 const matriz= [1, 2, 3];
3
4 for (let elemento of matriz) {
5   console.log(elemento); // 1, 2, 3
6 }
7
8 //for...in -> non-iterable object
9 const obj = { name: "Alice", age: 25 };
10
11 for (let key in obj) {
12   console.log(key, obj[key]); // name Alice, age 25
13 }
```

Console tab:

Output
1
2
3
name Alice
age 25

iterating through objects using iterable or non iterable version of for

Continue

8.- Loops

```
//continue will finish the actual iteration and move on to the next one
for (i = 0; i < 10; i++) {
    // if true, skip the remaining part of the body
    if (i % 2 == 0) continue;
    console.log(i);
}

//same effect as before, but better coding
for (i = 0; i < 10; i++) {
    if (i % 2) {
        console.log( i );
    }
}

//break/continue can't be used with ?
i=0;
(i<10) ? i++ : continue;
```

continue

Break

8.- Loops

```
//you can use break to exit the loop, while it is not recommended.  
//The best way to leave it is the condition  
let suma=0;  
//bad coding  
while (1){ //will repeat forever,  
    suma++;  
    if (suma==5) break;  
}  
  
//better  
suma=0;  
while (suma<5){  
    suma++;  
}  
  
//much better  
for (suma=0;suma<5;suma++);
```

break

Break labels

8.- Loops

```
//break only stops one loop so, in order to stop two or more, labels have to be used
salida: for (let i = 0; i < 3; i++) {
    for (let j = 0; j < 3; j++) {

        let input = prompt(`Value at coords (${i},${j})`, '');
        // if an empty string or canceled, then break out of both loops
        if (!input) break salida;

        console.log(`you have chosen ${i}${j}`);
    }
}
```

break labels

9.- Functions

1. Basic aspects

2. Advanced aspects

```
function add(num1, num2) {  
    // code  
    return result;  
}  
  
let x = add(a, b);  
// code
```



A diagram illustrating a function call. A blue arrow points from the identifier 'add' in the assignment statement 'let x = add(a, b);' to the opening brace of the function definition 'function add(num1, num2) {'. To the right of this arrow, the text 'function call' is written.

9.1.- Basic aspects of functions

```
function add(num1, num2) {  
    // code  
    return result;  
}  
  
let x = add(a, b);  
// code
```



A diagram illustrating a function call. A blue bracket labeled "function call" encloses the expression "add(a, b)". An arrow points from the variable "x" to this bracket.

1. Creating functions
2. Local and global variables
3. Arguments
4. Return values
5. Functions as expressions
6. Arrow functions
7. Callbacks

Variable scope

9.- Functions > 9.1.- Basic aspects

The screenshot shows a code editor with two tabs: 'script.js' and 'Console'. The 'script.js' tab contains the following code:

```
script.js ×

1 let adios="adios";
2 function hola(){
3     let adios="nos despedimos";
4     console.log (adios);
5 }
6 hola();
7
```

The 'Console' tab shows the output:

```
Console ×

nos despedimos
```

local scope has preference over global one

Arguments

9.- Functions > 9.1.- Basic aspects

The screenshot shows a code editor with two tabs: 'script.js' and 'Console'. The 'script.js' tab contains the following code:

```
script.js ×
1  function muestra_argumentos(nombre, saludo){
2      for (let key in arguments){
3          console.log (arguments[key]);
4      }
5      /* Alternative way of iterating
6      for (let i=0; i<arguments.length; i++){
7          console.log (arguments[i]);
8      }*/
9  }
10
11 muestra_argumentos("pepe", "hola");
```

The 'Console' tab shows the output:

```
Console ×
pepe
hola
```

A small note at the bottom right of the code editor says 'Iterating trough arguments'.

Arguments

9.- Functions > 9.1.- Basic aspects

script.js X

```
1  /*how to create a default value when an argument is not provided. Is a feature of modern JavaScript*/
2  function muestraCliente2(nombre, apellido="sin apellido") { //default value for arguments
3      nombre="el inconfundible "+nombre;
4      console.log(nombre+" "+apellido);
5  }
6
7  let nombres=["procopio","patrocinio"];
8
9  for (let i=0; i<nombres.length; i++)
10     muestraCliente2(nombres[i]);    //when missing an argument, its default value is used (defined in the function)
11
12
```

Console X

```
el inconfundible procopio sin apellido
el inconfundible patrocinio sin apellido
```

Default arguments in a function

Arguments as a value

9.- Functions > 9.1.- Basic aspects

```
script.js ×

1 //an argument is a copy of a global variable. Modifying it don't change global variable
2 ↘function muestraCliente(nombre, apellido) {
3   | nombre="el inconfundible "+nombre;
4   | console.log(nombre+" "+apellido);
5 }
6
7 let nombre="Perico";
8 muestraCliente(nombre, "Pérez");
9 console.log(nombre); //nombre variable won't be modified
10
```

Console ×

```
el inconfundible Perico Pérez
```

```
Perico
```

Primitive arguments are passed as value

Arguments as a reference

9.- Functions > 9.1.- Basic aspects

The screenshot shows a browser developer tools interface with two tabs: 'script.js ×' and 'Console ×'. The 'script.js' tab contains code demonstrating two functions: 'modificarObjeto' and 'reasignarObjeto'. The 'Console' tab shows the output of running this code, illustrating how objects are passed by reference.

```
script.js ×

1  function modificarObjeto(obj) {
2    | obj.propiedad = 'nuevo valor'; // Modifica el objeto original
3  }
4
5  let miObjeto = { propiedad: 'valor original' };
6  modificarObjeto(miObjeto);
7  console.log(miObjeto.propiedad); // 'nuevo valor' (cambia)
8
9
10 function reasignarObjeto(obj) {
11   | obj = { propiedad: 'otro valor' }; // Reasigna a un nuevo objeto, pero no afecta el original
12 }
13
14 let otroObjeto = { propiedad: 'valor original' };
15 reasignarObjeto(otroObjeto);
16 console.log(otroObjeto.propiedad); // 'valor original' (no cambia)

Console ×

nuevo valor
valor original
```

Objects are passed as reference

Returning values

9.- Functions > 9.1.- Basic aspects

script.js ×

```
1  function muestraCliente2(nombre, apellido="sin apellido") {  
2    nombre="el inconfundible "+nombre;  
3    console.log(nombre+" "+apellido);  
4  }  
5  
6  let nombres=["procopio","patrocinio"];  
7  
8  for (let i=0; i<nombres.length; i++)  
9    muestraCliente2(nombres[i]);    //when missing an argument, its default value is used (defined in the function)  
10
```

Console ×

```
el inconfundible procopio sin apellido  
el inconfundible patrocinio sin apellido
```

A default argument in a function

Returning values

9.- Functions > 9.1.- Basic aspects



The screenshot shows a code editor window with a dark theme. It displays two files: 'index.js' and 'buscador.js'. The 'index.js' file contains the following code:

```
function muestraCliente3(nombre, apellido=buscaApellido(nombre)) {
  //default value for arguments
  nombre="el inconfundible "+nombre;
  console.log(nombre+" "+apellido);
}

function buscaApellido(nombreBuscado){
  let nombres=[{nombre:"procopio",apellido:"máximo",
    {nombre:"patrocinio",apellido:"Sánchez"}];
  for (let i=0; i<nombres.length; i++){
    if (nombreBuscado==nombres[i].nombre){
      return nombres[i].apellido;
    }
  }
}

muestraCliente3("procopio");
```

The 'buscador.js' file contains the following code:

```
'el inconfundible procopio máximo'
```

A small button labeled 'Activar' is visible in the bottom right corner of the editor window.

A function as a default argument in a function

Returning values

9.- Functions > 9.1.- Basic aspects



The screenshot shows a code editor window with a dark theme. The code is written in JavaScript and demonstrates returning values from functions.

```
"use strict";
let usuario={nombre:"pepe",edad:"17"};
function compruebaMayoriaEdad(edad) {
    if (edad >= 18) {
        return true;
    } else {
        return false;
    }

    //alternative #1 (BEST)
    //return (usuario.edad>=18);

    //alternative #2
    //(usuario.edad>=18) ? return true : return false;
}

function garantizaAcceso(usuario){
    if (compruebaMayoriaEdad(usuario)){
        return(`acceso garantizado a ${usuario.nombre}`);
    }else{
        return(`acceso denegado a ${usuario.nombre}`);
    }

    //alternative
    /*let accion;
    compruebaMayoriaEdad(usuario) ? accion="garantizado" :
    accion="denegado";
    return (`Acceso a ${usuario.nombre} ${accion}`);*/
}

console.log(garantizaAcceso(usuario));
```

The output of the console.log statement is displayed in the bottom right corner of the editor window:

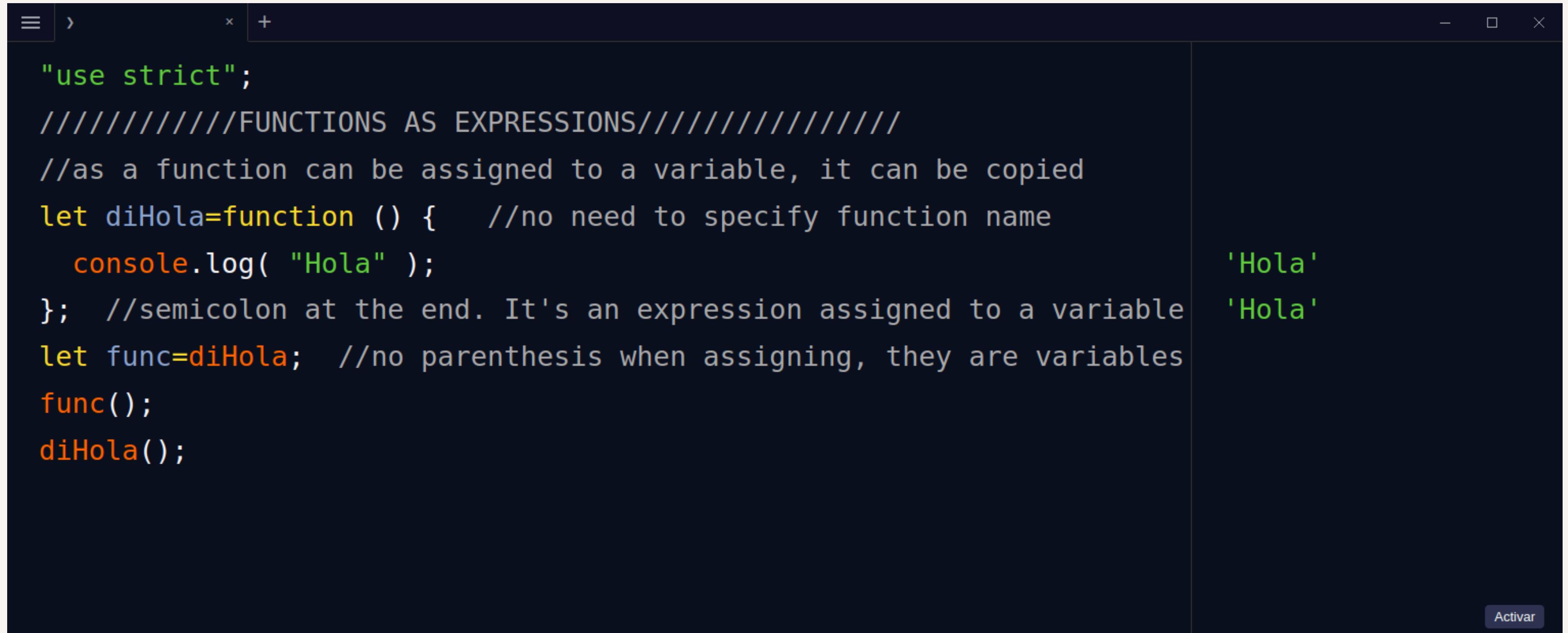
'acceso denegado a pepe'

Activar

a JavaScript function can return a value

Function as an expression

9.- Functions > 9.1.- Basic aspects



The screenshot shows a code editor window with the following code:

```
"use strict";
//////////FUNCTIONS AS EXPRESSIONS///////////
//as a function can be assigned to a variable, it can be copied
let diHola=function () { //no need to specify function name
    console.log( "Hola" );
}; //semicolon at the end. It's an expression assigned to a variable
let func=diHola; //no parenthesis when assigning, they are variables
func();
diHola();
```

The output on the right side of the editor shows the results of running the code:

```
'Hola'
'Hola'
```

A small button labeled "Activar" is visible in the bottom right corner of the editor window.

as JavaScript functions return a value, they can be seen as an expression and can be assigned to a variable or copied

Function as an expression

9.- Functions > 9.1.- Basic aspects

The screenshot shows a code editor window with a dark theme. It contains two examples of JavaScript code. The first example defines a function and uses it immediately:

```
//Another example
// program to find the square of a number
// function is declared inside the variable
let x = function (num) { return num * num };
console.log(x(4));
```

The output of this code is **16**, which is displayed to the right of the console log line.

The second example shows a function being assigned to a variable and then used:

```
// function can be used as variable value for other variables
let y = x(3);
console.log(y);
```

The output of this code is **9**, which is displayed to the right of the console log line.

Function can be used as variable value fot other variables

Activar

Function as an expression

9.- Functions > 9.1.- Basic aspects

The screenshot shows a browser developer tools console window with a dark theme. It contains two examples of JavaScript code. The first example, on the left, shows a standard function declaration:"use strict";
saluda("Procopio");
function saluda(nombre){
 console.log("hola "+nombre);
}The second example, on the right, shows a function expression:ReferenceError: diHola is not defined
'hola Procopio'

The output on the right shows a red error message 'ReferenceError: diHola is not defined' followed by the string 'hola Procopio'. This demonstrates that a function declared with 'function' can be called before it is defined, while a function expression cannot.

Function declared can be called before creating them while function as an expression can't

Activar

Function as an expression

9.- Functions > 9.1.- Basic aspects

```
"use strict"  
if (1){  
    function saluda(nombre){  
        console.log("hola "+nombre);  
    }  
}else{  
    function despidete(nombre){  
        console.log("adiós "+nombre);  
    }  
}  
saluda("pepe");
```

ReferenceError
: saluda is
not defined



```
"use strict";  
//this is the key. I define the variable outside  
the scope and, assign to the function inside  
let saluda, despidete;  
if (1){  
    saluda=function(nombre){  
        console.log("hola "+nombre);  
    }  
}else{  
    despidete=function(nombre){  
        console.log("adiós "+nombre);  
    }  
}  
  
saluda("pepe");
```

'hola pepe'

f saluda()

Declared functions are valid within scope they were declared at, while functions as an expression can be utilized outside if declared

Arrow functions

9.- Functions > 9.1.- Basic aspects

The screenshot shows a code editor interface with two tabs: 'script.js' and 'Console'. The 'script.js' tab contains the following code:

```
script.js ✘

1 //similar to function expressions, but specifying the returning value
2 let suma=(a,b) => a+b;
3 let saluda=(nombre)=> console.log("yo te saludo "+nombre);
4 let despidete=()=>console.log("adiós");
5
6 let edad=prompt("¿cuál es tu edad?");
7 let vasPreso=(edad>=18) ?
8   () => console.log("vas preso") :
9   () => console.log("al correccional");
10
11 saluda();
12 vasPreso();
```

The 'Console' tab shows the output of the script:

```
Console ✘
yo te saludo undefined
al correccional
```

A small note at the bottom of the editor says: '?' : syntax can be used

? : syntax can be used

Arrow functions

9.- Functions > 9.1.- Basic aspects

```
script.js ×

1 //for multiline functions, curly brace and return statement must be used
2 let resta=(a,b)=>{
3   let resultado=a-b;
4   return (resultado); //could've been done in a single line, but for demonstration purposes
5 }
6
7 let suma=(a,b)=>{
8   return (a+b);
9 }
10
11 console.log (resta(4,2));
12 console.log(suma(4,2));
13

Console ×

2
6
```

Multi line arrow functions must use curly braces and return

Callbacks

9.- Functions > 9.1.- Basic aspects

```
function ask(question, yes, no) {  
  if (confirm(question)) yes() //confirm shows up a window asking accept or cancel  
  else no();  
}  
  
function showOk() {  
  console.log( "You agreed." );  
}  
  
function showCancel() {  
  console.log( "You canceled the execution." );  
  'You canceled the execution.'  
}  
  
// usage: functions showOk, showCancel are passed as arguments to ask  
ask("Do you agree?", showOk, showCancel);
```

Functions can be passed as an argument to another function

Callbacks

9.- Functions > 9.1.- Basic aspects

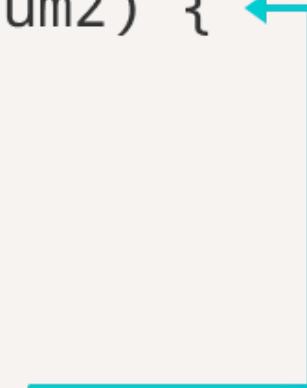
```
//Another way to express the last code (not recommended though)
function pregunta(question, yes, no) {
  if (confirm(question)) yes() //confirm shows up a window asking accept or cancel
  else no();
}

pregunta(
  "Agree?",
  function() {console.log("estás de acuerdo");},
  function () { console.log ("no estás de acuerdo") }           'no estás de acuerdo'
);
```

Another way to express functions callbacks

9.2.- Advanced aspects of functions

```
function add(num1, num2) {  
    // code  
    return result;  
}  
  
let x = add(a, b);  
// code
```

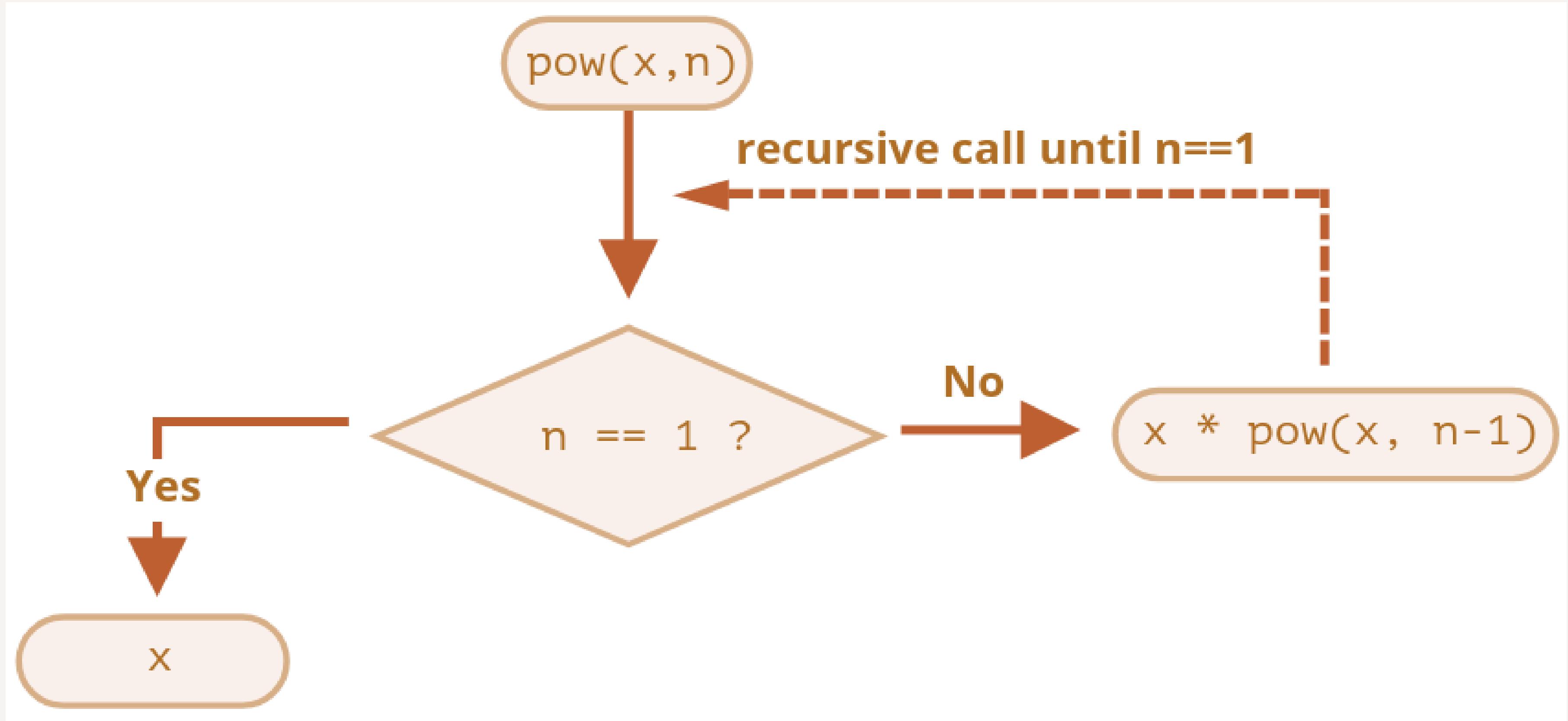


A diagram illustrating a function call. A blue bracket labeled "function call" connects the identifier "add" in the assignment statement "let x = add(a, b);" to the opening brace of the function definition "function add(num1, num2) {". A blue arrow points from the "a" in "add(a, b)" to the "num1" parameter in the function definition.

1. Recursivity
2. Rest parameters
3. Spread syntax
4. Autoexecutable functions
5. Nested functions
6. Scope
7. Closure

Recursivity

9.- Functions > 9.2.- Advanced aspects



for a algorithm to be expressed as recursive, it is needed a base case and a problem that can be decomposed in subproblems

Recursivity

9.- Functions > 9.2.- Advanced aspects

```
function pow(x, n) {  
    let result = 1;  
  
    // multiply result by x n times in the loop  
    for (let i = 0; i < n; i++) {  
        result *= x;  
    }  
  
    return result;  
}  
  
alert( pow(2, 3) ); // 8
```

```
function pow(x, n) {  
    if (n == 1) {  
        return x;  
    } else {  
        return x * pow(x, n - 1);  
    }  
}  
  
alert( pow(2, 3) ); // 8
```

REST parameters

9.- Functions > 9.2.- Advanced aspects

The screenshot shows a code editor with two tabs: "script.js" and "Console". The "script.js" tab contains the following code:

```
1  "use strict";
2
3  //Rest parameters
4  //allows to pass an undetermined number of parameters as an array. Must be placed at the end
5  function sumaTodo(num1, num2, ...numeros){
6      let acumulado=num1+num2;
7      for (let num of numeros){
8          acumulado+=num;
9      }
10     return acumulado;
11 }
12
13 //palabras is an array containing all elements passed as arguments
14 let concatena2 = (...palabras) => {
15     let resultado="";
16     for (let palabra of palabras){
17         resultado+=palabra;
18     }
19     return resultado;
20 }
21
22 console.log (sumaTodo(1,2,3,4,5,6,7,8));
23 console.log (concatena2("a", "b", "c", "d"));
```

The "Console" tab shows the output of the code execution:

```
36
abcd
```

REST parameters allow to pass an indetermined number of arguments

Spread syntax

9.- Functions > 9.2.- Advanced aspects

The screenshot shows a browser's developer tools interface with two tabs: 'script.js' and 'Console'. The 'script.js' tab contains the following code:

```
script.js ×

1 //=====Spread syntax=====/
2 let arr = [3, 5, 1];
3 let arr1 = [1, -2, 3, 4];
4 let arr2 = [8, 3, -8, 1];
5
6 //an array can't be passed to Math.max, as it expects a list of items.
7 //Spread syntax “expands” an iterable object arr into the list of arguments.
8 console.log( Math.max(1, ...arr, ...arr1, 2, ...arr2, 25));
```

The 'Console' tab shows the output: 25. Below the code, a note states: "Spread syntax expands an iterable object".

Spread syntax expands an iterable object

Autoexecutable functions

9.- Functions > 9.2.- Advanced aspects

```
script.js ×

1  (function() { console.log("hola mundo") }) ();
2
3  ( function(quien){
4    |   console.log("hola " + quien);
5  })("mundo");
```

```
Console ×
```

```
hola mundo
```

```
hola mundo
```

Nested functions

9.- Functions > 9.2.- Advanced aspects

script.js ×

```
1  function externa() {  
2      console.log("Soy la función externa");  
3  
4      function interna() {  
5          console.log("Soy la función interna");  
6      }  
7  
8      interna(); // Llamamos a la función interna desde dentro de la función externa  
9  }  
10 //interna(); //error: unknown function  
11 externa(); // Llamada a la función externa
```

Console ×

Soy la función externa

Soy la función interna

Nested functions. Inner function is invisible outside

Nested functions

9.- Functions > 9.2.- Advanced aspects

script.js ×

```
1 //a function created within another function
2 //inner function is invisible outside and can use outer variables
3 function saludador(quien){
4     return function(){ //creación de la funcion anónima a retornar. No hace falta nombre
5         console.log("hola " + quien);
6     }
7 }
8 var saluda = saludador("mundo");
9 saluda(); //hola mundo
```

Console ×

hola mundo

It is possible to return a nested function. Thus, it is summoned when calling the external one

Scope

9.- Functions > 9.2.- Advanced aspects

script.js X

```
1 let mensaje1="hola don pepito",
2     | mensaje2="hola don josé";
3
4 function muestraMensaje() {
5     let mensaje1 = "Saludos desde dentro de la función!"; // If a local variable with the same name is defined, global variable can't be used
6     if (1){
7         let mensaje1="mensaje 1 dentro del if";
8     }
9     console.log (mensaje1, mensaje2);
10    mensaje2="cambio el mensaje dentro de la función"; //a global variable can be accessed and modified inside a function
11 }
12
13 console.log ("muestro variables globales antes de llamar a la función:", mensaje1, mensaje2); //showing global message variable
14 muestraMensaje();
15 console.log ("muestro variables globales después de llamar a la función:", mensaje1, mensaje2); //showing global message variable
```

Console X

muestro variables globales antes de llamar a la función: hola don pepito hola don josé

Saludos desde dentro de la función! hola don josé

muestro variables globales después de llamar a la función: hola don pepito cambio el mensaje dentro de la función

Use of global, local and block scope in a function

Scope

9.- Functions > 9.2.- Advanced aspects

The screenshot shows a code editor window titled "script.js" and a browser's developer tools window titled "Console". The code in "script.js" is as follows:

```
1 let globalVar = 'Global';
2 function exampleFunction() {
3     let localFuncVar = 'Local a la función';
4     console.log(globalVar);
5     console.log(localFuncVar);
6     //console.log(localIfVar); //Error. It's not defined
7     if (1){
8         let localIfVar="Local al if";
9         console.log(globalVar);
10        console.log(localFuncVar);
11        console.log(localIfVar);
12    }
13 }
14
15 exampleFunction();
16 console.log(globalVar);
17 //console.log(localFuncVar); //Error. It's not defined
18 //console.log(localIfVar); //Error. It's not defined
```

The "Console" tab displays the following output:

Scope	Output
Global	Global
Local a la función	Local a la función
Global	Global
Local a la función	Local a la función
Local al if	Local al if
Global	Global

A note at the bottom of the code editor says "Global, function and block scope".

Scope

9.- Functions > 9.2.- Advanced aspects

script.js ×

```
1 let x = 1;
2
3 function func() {
4     console.log(x); // Error: No se puede acceder a 'x' antes de inicializarla
5     let x = 2;
6 }
7 func();
8
```

Console ×

- ▶ ReferenceError: can't access lexical declaration 'x2' before initialization
func@https://preview-javascript.playcode.io/ line 19 > injectedScript:68:15
@https://preview-javascript.playcode.io/ line 19 > injectedScript:72:1
mn@https://preview-javascript.playcode.io/ line 6 > injectedScript:19:5455

Error accessing a local variable not defined yet

Scope

9.- Functions > 9.2.- Advanced aspects

```
script.js ×

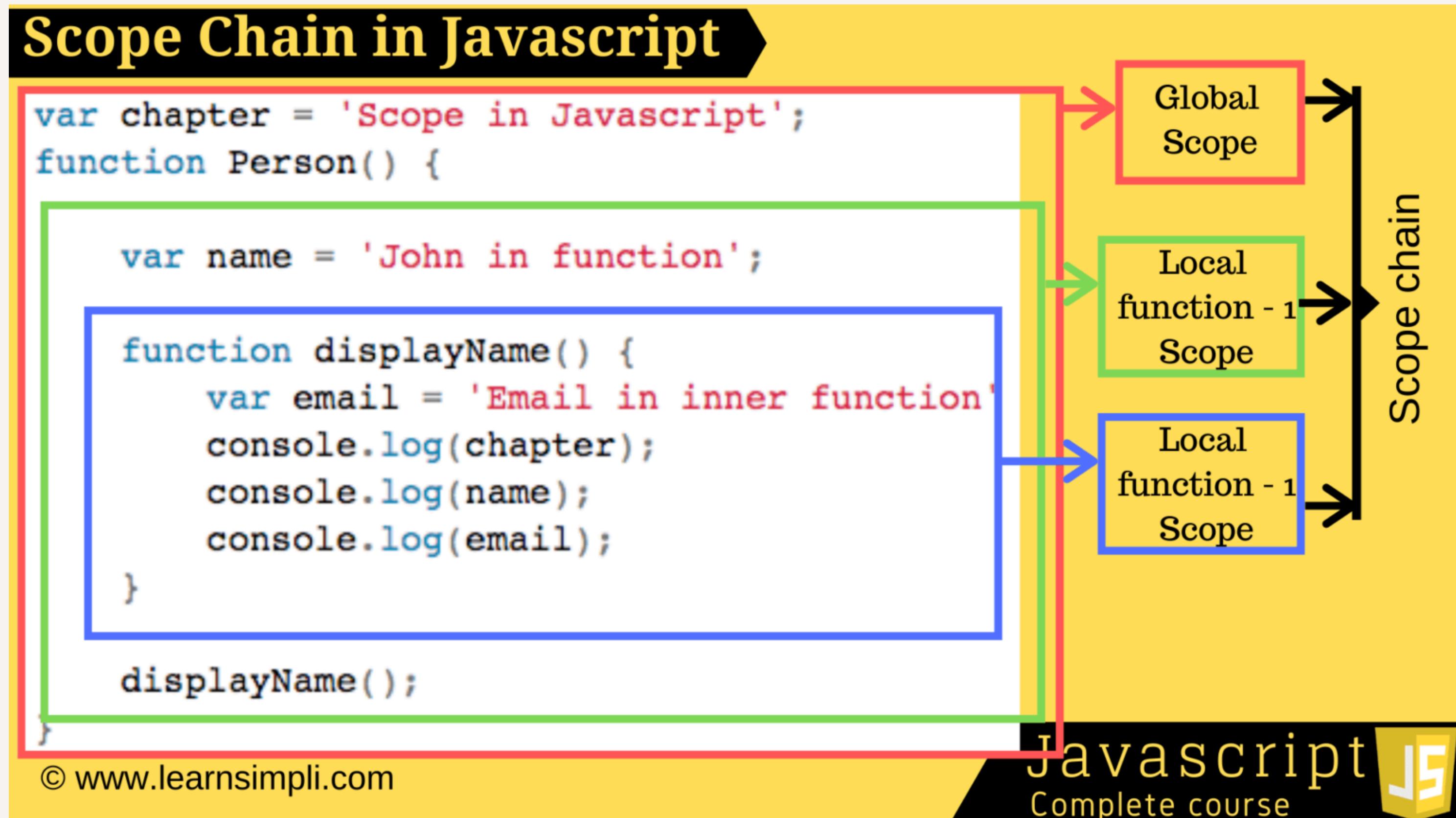
1  "use strict";
2
3  let frase = "hola";
4
5  if (1) {
6    let usuario = "Patrocinio";
7    function saluda() {
8      console.log(` ${frase}, ${usuario}`);
9    }
10 }
11
12 saluda(); //Saluda is unknown outside the "if" block
```

```
Console × ▶ ...
▶ TypeError: saluda is not a function
@https://preview-javascript.playcode.io/ line 19 > injectedScript:74:1
mn@https://preview-javascript.playcode.io/ line 6 > injectedScript:19:5455
```

Functions, as variables, can only be used within their scope

Scope and closure

9.- Functions > 9.2.- Advanced aspects



Scope and closure

9.- Functions > 9.2.- Advanced aspects

The screenshot shows a code editor interface with two tabs: 'script.js' and 'Console'. The 'script.js' tab contains the following JavaScript code:

```
1  function creaUsuario() {  
2      let nombre= "Pepe";  
3  
4      return function() {  
5          return nombre;  
6      };  
7  }  
8  
9  let nombre = "Manolo";  
10  
11 let usuario = creaUsuario(); // crea una función  
12 console.log(usuario()); // Pepe
```

The 'Console' tab shows the output: 'Pepe', indicating the value returned by the nested function.

Closure: Nested functions remember their lexical environment when they was created. It allows functions to remember their state

Scope and closure

9.- Functions > 9.2.- Advanced aspects

The screenshot shows a code editor window titled "script.js" and a browser's developer tools window titled "Console".

script.js:

```
1 function persona(aux){  
2     let nombre=aux;  
3     return function(){  
4         return nombre;  
5     }  
6 }  
7  
8 let persona1=persona("pepe");  
9 let persona2=persona("manolo");  
10 //same variable, different value  
11 console.log(persona1());  
12 console.log(persona2());
```

Console:

```
pepe  
manolo
```

A horizontal dashed line separates the code from the output.

Same variable name, different value due to scope