TUP | UTN - FRT

Material de apoyo a las clases.

JavaScript: Elementos básicos

JavaScript

JavaScript es un lenguaje dinámico que podemos usar tanto para desarrollar las interface de usuario como los servidores.

Identificadores

Los identificadores en JS se forma con una secuencia de letras, dígitos, "$" y "_"
No puede comenzar un digito ni formar una palabra reservada.

Convención de nombres

Se usa todo minúsculas en las propiedades de *html*
Se usa minúsculas y las palabras separadas por guion en las propiedades de *css*
En JavaScript

- Se usa mayúsculas y las palabras separadas por guion bajo en las constantes.
- Se usa minúsculas con la primera letra a partir de las segunda palabras en mayúsculas para variables y funciones.
- Se usa minúsculas con la primera letra de cada palabra en mayúsculas para las clases y constructores.

```
// html
<button onclick="cambiarColor()" style="background-color:red"> </button>
// css
button { background-color: red; }

//javascript
let primerNombre = "Juan"     // Variables y funciones
class PersonaFisica { ... }     // Clases y constructores
const COLOR_FONDO = "red" // Constantes

function cambiarColor(){
        document.body.style.backgroundColor = COLOR_FONDO
        // tambien se puede poner
```

```
        document.body.style = "background-color:red"
}
```

Declaración de variables

Declarar una variable definir un nombre para ser usado para almacenar valores.

Existe tres formas de declarar una variable:

- **var**: define una variable cuyo alcance el la función, se puede declarar varias veces, es la forma antigua NO recomendada.
- **let**: define una variable cuyo alcance es el bloque
- **const**: define una variable cuyo valor no va a cambiar, tiene alcance de bloque.

```
var nombre
let apellido
const COLOR

typeof(nombre) //> 'undefined'
```

Las variables al declarar no tiene un valor asignado así que se las considera 'undefined'.

Cuando se asigna un valor a una variable se define el tipo de datos. Un tipo de datos define que valores pueden almacenarse en la variable y que operaciones se pueden realizar.

JS es un lenguaje dinámico, se puede cambiar el tipo de una variable en cualquier momento con una simple asignación.

Se pueden declarar múltiples variables separándolas por ","
También se pueden asignar un valor en la declaración con el signo "="

```
let valor = "Nombre"
typeof(valor) //> 'string'

valor = 100
typeof(valor) //> 'number'

let a = 10, b = 20, c = 30 // Declara y define 3 variables. a, b y c

var variable = "valor"
// equivale a
var variable
variable = "valor"
```

El alcance (cuando se puede usar) de una variable depende si se usa var o let.

```
var x = 10
if(true){
        var x = 20 // vuelve a definir la misma variable
}

console.log(x) //> 20
var y = 10
if(true){
        let y = 20 // Crear una nueva variable local dentro de {}
}
console.log(y)  //> 10 (se usa la 'y' definida dentro de "{}")

let z = 10
if(true){
        z = 20;
}
console.log(z)  //> 20 (se usa la 'z' definida arriba)
```

---

numeros

Las variables pueden almacenar numeros y realizar cálculos sobre ellas.
JavaScript no diferencia entre numeros enteros o numeros flotantes (con punto decimal)
Los numeros siempre empiezan con un digito. Hay distintas representaciones o forma de expresar un número que el valor almacenado es siempre un número real mas allá de la forma de ingresarlos. Cuando las operaciones necesitan se aplicadas sobre numeros enteros (operaciones de bit) se hace una conversión interna en forma transparente.

Como todos los numeros se guardan igual no hay ninguna ventaja en usar enteros en lugar de número decimales, solo una cuestión de comodidad en el uso.

```
let a = 1000           // Entero o integer
let b = 1000.3         // Número real, flotante o float (lleva "." decimal)
let c = 1e3            // Notación cientifica (1 segido de 3 ceros)
let d = 0x3E8          // Hexadecimal  (p.e. colores en CSS)
let e = 0b001111101000 // Binarios
let f = 10_000_000     // Se puede usar "_" para facilitar lectura

// typeof(a), typeof(b), ... , typeof(f)  //> Todos son "number"
```

Operadores

Sobre los numeros se pueden realizar las operaciones matemáticas básicas para las cuales hay definidos operadores específicos.

Tienen el comportamiento tradicional de las matemáticas en lo que se refiere en el orden de calculo cuando las expresiones involucran varios operadores. Los paréntesis se evalúan primero (desde adentro hacia afuera), luego la potencia, luego los productos y por ultimo las sumas. Si la prioridad es la misma se evalúa de izquierda a derecha (asociatividad izquierda)

```
let suma = 1 + 2          //> 3
let producto = 2 * 3      //> 6
let mixto = 1 + 2 * 3     //> 7 ⟹ 1 + (2 * 3)
let division = 8 / 4 / 2 //> 1 (Se realizan de izquierda a derecha)
let resto = 10 % 3        //> 1
let potencia = 2 ** 3     //> 8 ⟹ 2 * 2 * 2
```

También se pueden usar operadores de comparación, <, >, <=, >=, == , !=

Los comparadores de comparación toma dos numeros y retorna un booleano, tiene menor prioridad que los operadores aritméticos

```
let esMenor = 10 < 20  //> true
let esMayor = 10 > 20  //> false
let esMenorIgual = 10 ≤ 20  //> true

let precio = 100
let cantidad = 10

let enPresupuesto = precio * cantidad ≤ 1000  //> true | primero * y luego
≤
```

funciones

Las variables de tipo numérico tiene una serie de funciones para usar.
Alguna se acceden con el "." como es el caso del toString() otras está en el modulo Math.

```
let edad = 28
let precio = 1234.56

// funciones para formatear numeros
edad.toString()     //> "28"
edad.toString(2)    //> "00011100" | 50 en binario
edad.toString(16)   //> "1C" | 50 en hexadecimal
edad.toFixed(2)     //> "28.00" | 28 con dos decimales
```

```
precio.toLocaleString() //> "1.234,56" | Convierte el número en formato
local

// Libreria Math
Math.PI             //> 3.141592653589793
Math.abs(-5)        //> 5

Math.floor(5.9)     //> 5
Math.ceil(5.1)      //> 6
Math.round(5.1)     //> 5

Math.random()       //> 0.12345678901234567 | Número aleatorio entre 0 y 1
Math.sin(Math.PI)   //> 1.2246467991473532e-16 | Seno de PI
Math.cos(Math.PI)   //> -1 | Coseno de PI

Math.min(1, 2, 3)   //> 1
Math.max(1, 2, 3)   //> 3

let a = parseInt("100")     //> 100
let b = parseFloat("100.2") //> 100.2
```

Conversión

JavaScript es un lenguaje débilmente tipado, esto significa que intenta convertir los datos para poder realizar las operaciones cuando las misma no están permitida en el tipo original.

En el caso que no pueda convertir una tipo a número lo convertirá en "NaN" (Not an Number) para indicar que no puedo pero no fallará, es decir no dará un error.

Hay otros caso que normalmente darían error en otros lenguajes pero que en JS no. La división por cero, por ejemplo.

La función Number() intenta convertir cualquier tipo a Number en forma explicita y es la que se usa internamente cuando hay operaciones númericas (como ser "/" o ">") que necesitan de un número para funciónar.

```
let a = 100 / 0    //> Infinity
let b = 0 / 0      //> NaN

let c = 100 / 'a'  //> NaN
let d = 100 / '2'  //> 50       | Convirtio "2" a 2

let e = "100" + 5  //> "1005"   | Concatenacion (convirtio 5 a "5")
let f = "100" - 5  //> 95       | Resta (convirtio "100" a 100)

let g = 100 / NaN  //> NaN      | NaN es un valor especial "Not a Number"
let h = 100 + NaN  //> NaN      | Cualquier operacion con NaN resulta en NaN
```

```
let i = Number("100")
let j = Number("100a")  //> NaN
let k = Number(true)    //> 1
let l = 100 + true      //> 101     | Convirtio true a 1
let m = Number([])          //> 0
let n = Number([1])         //> 1
let o = Number([1, 2, 3])   //> NaN

let p = Number({})          //> NaN
```

Por ejemplo, no se puede sumar un nuúmero y una cadena, en esta caso hay dos posibilidades, o usar "+" de las cadenas es decir la concatenación, o usar el "+" de los numeros o sea la suma. Como siempre se puede convertir un número a una cadena, pero no siempre se puede convertir una cadena en un número elige usar la concatenación por lo tanto convertirá el número a cadena antes de juntarlos.

Cadena

Las cadenas en JS son una secuencia de caracteres. Los caracteres son los letras y símbolos que usamos para comiscarnos, en este caso incluye todo los símbolos de todos los idiomas que están registrados en Unicode.

Se las reconoce porque comienzan y terminan con comillas, ya sean doble, simples o invertidas.
Las comillas dobles aceptar comillas simples en su interior y las comillas simple aceptar comillas dobles en su interior.

Para agregar caracteres especiales se debe prefijar con "". Por ejemplo \r comienza una nueva línea, "\t" salta espacios horizontales. etc.

Las comillas invertidas es la forma moderna de escribir cadenas, es la mas poderosa porque permite multilínea e interpolación de expresiones.

```
let nombre = "Juan"             // Comillas dobles
let apellido = 'Perez'          // Comillas simples
let domicilio = `Calle 123`     // Comillas invertidas

let nombreCompleto = nombre + ' ' + apellido //> "Juan Perez" |
Concatenación

nombreCompleto = `${nombre} ${apellido}`     //> "Juan Perez" |
Interpolación

let expresion = `1 + 2 = ${1 + 2}`           //> "1 + 2 = 3" | Interpolación
de expresión
```

```
let html = `
    <div>
        <h1>${nombreCompleto}</h1>
        <p>${domicilio}</p>
    </div>
`                                          //> HTML con interpolación
```

Operadores

Las cadenas acepta el operador "+" como los numeros pero en este caso lo que hace es concatenar (o juntar) dos cadenas.
Equivale a la función .contact()

```
let nombre = "Juan"
let apellido = "Perez"

let nombreCompleto = nombre + " " + apellido     // Con operador "+"
nombreCompleto = nombre.concat(" ", apellido)    // Con la función concatenar

nombreCompleto = `${nombre} ${apellido}`             // Con interpolación de
string
```

funciones

Las cadenas soporta una gran cantidad de funciones.

```
let nombre = "Juan"

nombre.length //> 4    | Cantidad de caracteres

// Acceso a caracteres
nombre[0]                       //> "J"   | Primer caracter
nombre[1]                       //> "u"   | Segundo caracter
nombre[nombre.length-1]         //> "n"   | Ultimo caracter

nombre.at(0)                    //> "J"   === nombre[0]
nombre.at(1)                                    //> "u"   === nombre[1]
nombre.at(-1)                                   //> "n" ===
nombre[nombre.length-1]
nombre.at(-2)                                   //> "a" ===
nombre[nombre.length-2]

// Extraer una parte de la cadena
nombre.slice(1)                                 //> "uan"   | Desde el
caracter 1 hasta el final
nombre.slice(1, 3)                              //> "ua"   | Desde el
```

```
caracter 1 hasta el 3
nombre.split("")                                  //> ["J", "u", "a", "n"] |
Convertir a array
"uno,dos,tres".split(",")                  //> ["uno", "dos", "tres"] | Separar
por coma

// Buscar una subcadena
nombre.indexOf("a")              //> 2   | Posicion de la primera "a"
nombre.indexOf("x")              //> -1  | No se encontro "x"

nombre.includes("a")             //> true  | Existe "a" en la cadena
nombre.includes("x")             //> false | No existe "x" en la cadena

nombre.startsWith("J")           //> true  | Empieza con "J"
nombre.endsWith("n")             //> true  | Termina con "n"

// Convertir a mayusculas o minusculas
nombre.toUpperCase()             //> "JUAN"   | Mayusculas
nombre.toLowerCase()             //> "juan"   | Minusculas

// Reemplazar una subcadena
nombre.replace("a", "o")         //> "Juon" | Reemplaza "a" por "o"
nombre.replace(/aeiou/g, "x")    //> "Jxxn" | Reemplaza las vocales por "x"
```

Conversión

Es interesante que todo objetos de JS se pueden convertir a cadenas para lo que tiene la función toString(). No todas las conversiones tienen sentido pero son posibles de realizar.

Cuando usamos el operador "+" con una cadena y un número siempre convierte el número en cadena. No usa la operación matemática "+". En cambio si usara el operador "-" ya que el mismo no esta definido para las cadenas trataría de convertir la cadena en número.

```
let expresion = "100" + "200";       //> 100200
expresion = "100".concat("200"); //> 100200

expresion = "100" + 200; //> 100200 | Usa concatenación en lugar de la suma
expresion = 100 + "200"; //> 100200

// Conversión explicita
let a = String(10)                                  //> "10"
let b = String(true)                        //> "true"
let c = String([1, 2, 3])                   //> "1,2,3"
let d = String({a: 1, b: 2})     //> "[object Object]"
let e = String(null)                        //> null
let f = String(undefined) //> "undefined"
```

Booleano

Los tipos booleanos son el resultado de una comparación.
Puedo tomar solo dos valores las palabras reservadas true y false según la comparación sea verdadera o falsa.

```
let a = true    //> true
let b = !true   //> false  | "!" es el operador de negación
let c = !!true  //> true   | "!!" es el operador de doble negación
```

Lo especial de JS es que realiza una conversión automática haciendo que cualquier valor distinto de "false", 0, "", null, undefinded se considerada true

```
let a = Boolean(0)  //> false
let b = Boolean(1)  //> true
let c = Boolean(2)  //> true

let d = Boolean('')  //> false
let e = Boolean(' ')  //> true
let f = Boolean('a')  //> true
let g = Boolean('0')  //> true

let h = Boolean(null)  //> false
let i = Boolean(undefined)  //> false

let j = Boolean([])  //> true
let k = Boolean({})  //> true
let l = Boolean(NaN)  //> false

let m = 1 == true //> true
let n = 2 == true //> false     | Convierte el booleano a número
let o = 0 == false //> true

let p = "true" == true //> false | Convierte a número ⟹ NaN == 1 ⟹ false
```

Operadores

Los tipo booleanos aceptar los operadores lógicos && (and), || (or) y ! (not).

El operador && retorna true si todos los valores son true. En realidad el comportamiento es un poco mas especifico, retorna el ultimo elemento verdadero o false.

El operadores || retorna true si algún elemento retorna true. En realidad el comportamiento exacto es retornar el primer elemento que sea false o true.

El operador ! invierte el resultado. Si se usa dos veces !! convierte a bolean cualquier valor.

JS usa lo que se conoce como "cortocircuito" es decir una expresión lógica solo se evalúa hasta que se puede conocer su resultado. En un && hasta el primer false, en un || hasta el primer true. Debe entenderse que la conversión de tipos solo se hace a los fines de la evaluación de la expresión no en el retorno de los resultados.

Los operadores booleanos tienen menor presedencia que la comparacion. A su vez, && se evalua antes que ||

```
true && true     //> true
true && 10       //> 10 ⇒ true | 10 se convierte a true para evaluar pero
retorna 10

let x = 10
true && x        //> x sin importar su valor porque la primera parte es true
false && x       //> false sin importar el valor de x porque la primera parte
es false

x = false
x && console.log('Hola')    //> console.log no se ejecuta porque x es false
x = true
x && console.log('Hola')    //> Hola ⇒ console.log se ejecuta porque x es
true y se evalua la segunda parte (undefined ⇒ false)

true  || false  //> true
false || x      //> x | x porque la primera parte es falsa
true  || x      //> true | true porque la primera parte es verdadera

true || console.log("Hola") //> false | false porque la primera parte es
true

// Asignacion de valores por defecto
let nombre

nombre = nombre || "Desconocido"  //> "Desconocido" | nombre es undefined,
por lo que se asigna el valor por defecto
nombre ||= "Desconocido"          //> "Desconocido" | forma abreviada

nombre = "Juan"
nombre ||= "Desconocido"          //> "Juan" | nombre ya tiene un valor, por
lo que no se asigna el valor por defecto
```

Conversión

Cuando JS espera un valor booleano (como en un if, while, for, etc) hacer una conversión automática.

Los valores false, 0, "", null, undefined y NaN se convierten a false.
Todos los demas valores se convierte a true.

```
Boolean(false)          //> false
Boolean(true)           //> true
Boolean(0)              //> false
Boolean(1)              //> true
Boolean("")             //> false
Boolean(" ")            //> true
Boolean([])             //> true
Boolean({})             //> true
Boolean(null)           //> false
Boolean(undefined)      //> false

Boolean("true")         //> true
Boolean("false")        //> true
```

Array

Los array se usan para representar una colección ordenadas de valores. Esta colección puede ser de cualquier tipo. Normalmente los elementos suelen ser del mismo tipo que esto no es necesario.

Los array se reconocen porque estan encerrado entre "[" y "]" y sus elementos estan separados por ",". Dos comas consecurivas indica que el elemento se omitio, excepto una coma la final que es simplemente ignorada.

```
let numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

numeros.length      //> 10  | Cuanto elementos tiene
numeros[0]          //> 1   | Accede al primer elemento (base 0)
numeros[9]          //> 10  | Accede al último elemento
numeros[numeros.length - 1]  //> 10  | Accede al último elemento
numeros[100]        //> undefined  | No hay un elemento en la posición 100

numeros.at(0)       //> 1   | Accede al primer elemento (base 0)
numeros.at(-1)      //> 10  | Accede al último elemento | Equivale a
numeros[numeros.length - 1]
numeros.at(-2)      //> 9   | Accede al penúltimo elemento | Equivale a
numeros[numeros.length - 2]

numeros[2] = 100    // Cambia el valor del elemento en la posición 2 (3er
elemento)
numeros[numeros.length] = 200 // Agrega un elemento al final del arreglo
numeros[20] = 300   // Agrega un elemento en la posición 20 (rellena con
undefined)
```

```
numeros             //> [1, 2, 100, 4, 5, 6, 7, 8, 9, 10, 200,,,,,,,,,,
300]
```

Los array pueden contener cualquier tipo de datos, incluso otros objetos. Para hacer una matriz de usa array dentro de array.

```
let mixto = [1, "Hola", [10, 20], {nombre: "Juan"}]
mixto[0]  //> 1
mixto[1]  //> "Hola"
mixto[2]  //> [10, 20]
mixto[2][0]  //> 10

let cuadrada = [
    [10, 20],
    [30, 40],
]

cuadrada[0]  //> [10, 20]
cuadrada[0][0]  //> 10
cuadrada[0][1]  //> 20
cuadrada[1]  //> [30, 40]
cuadrada[1][0]  //> 30
cuadrada[1][1]  //> 40
```

funciones

Existen muchas funciones que se aplican a los array, hay funciones para agregar o quitar elementos.

```
let numeros = [10, 20, 30, 40]

numeros.push(1000)      // adiciona um elemento no final do array
numeros                 //> [10, 20, 30, 40, 1000]
numeros.unshift(2000)   // adicion 2000 al início del array
numeros                 //> [2000, 10, 20, 30, 40, 1000]
numeros.pop()           //> 1000 | elimina el último elemento del array
numeros                 //> [2000, 10, 20, 30, 40]
numeros.shift()         //> 2000 | elimina el primer elemento del array
numeros                 //> [10, 20, 30, 40]

numeros.length = 3     // elimina los elementos que sobran
numeros               //> [10, 20, 30]
numeros.length = 5     //> [10, 20, 30, <2 empty items>] | adiciona
elementos vacios
numeros.length = 0     // elimina todos los elementos
```

```
// Sacar partes de un array

let numeros = [10, 20, 30, 40, 50]
numeros.slice(2, 4)   //> [30, 40]      | saca los elementos de la posición
2 a la 4 (sin incluir la 4)
numeros.slice(2)      //> [30, 40, 50]  | saca los elementos de la posición
2 hasta el final
numeros.slice(-2)     //> [40, 50]      | saca los últimos 2 elementos
numeros.slice(-3, -1) //> [40, 50]      | saca los elementos de la posición
-3 a la -1 (sin incluir la -1)
numeros.slice()       //> [10, 20, 30, 40] | copia el array

// Eliminar elementos de un array
let numeros = [10, 20, 30, 40, 50]
numeros.splice(2, 2) //> [30, 40] | elimina los elementos de la posición 2 a
la 2
numeros              //> [10, 20, 50]
numeros.splice(2, 0, 30, 40) //> [] | no elimina ningún elemento, pero añade
los elementos 30 y 40 en la posición 2
numeros              //> [10, 20, 30, 40, 50]

numeros.splice(2, 1, 35) //> [30] | reeemplaza el elemento de la posición 2
por 35
numeros                  //> [10, 20, 35, 40, 50]

numeros.reverse()        //> [50, 40, 35, 20, 10] | invierte el array
numeros.sort()           //> [10, 20, 35, 40, 50] | ordena el array (pero
como strings)

[2, 10, 3, 20].sort()    //> [10, 2, 20, 3] | ordena como strings

// Buscar elementos en un array
numeros.includes(20)     //> true | busca el elemento 20
numeros.includes(100)    //> false | busca el elemento 100

numeros.indexOf(20)      //> 1    | busca la posición del elemento 20
numeros.indexOf(100)     //> -1   | si no lo encuentra devuelve -1

numeros.find(x => x > 20) //> 35 | busca el primer elemento que cumple la
condición
numeros.find(x => x > 100) //> undefined | si no lo encuentra devuelve
undefined

numeros.findIndex(x => x > 20) //> 2 | busca la posición del primer elemento
que cumple la condición
numeros.join("|")              //> "10|20|35|40|50" | convierte el array en
un string separado por "|"
"10|20|35|40|50".split("|")   //> ["10", "20", "35", "40", "50"] | convierte
el string en un array separado por "|"
```

Existen ademas muchas funciones para programación funcional pero que veremos cuando veamos funciones.

Objetos

Los objetos en js nos permite almacenar una coleccion de valores identificados por una clave.
Esto permite que valores que estan conceptuamente unidos (como el nombre y apellido de una persona) se manejen como una unidad.

Se identifican porque comienza y termina con una "{" "}" en el contexto de una asignacion de valor.

```
// Multiples valores relacionados
let nombre = "Juan"
let apellido = "Perez"

function nombreCompleto(nombre, apellido) {
    return `${nombre} ${apellido}`
}

nombreCompleto(nombre, apellido) //> Juan Perez

// Lo mismo pero con un objeto
let persona = {nombre: "Juan", apellido: "Perez"}
function nombreCompleto(persona) {
    return `${persona.nombre} ${persona.apellido}`
}

nombreCompleto(persona) //> Juan Perez

persona = {"nombre": "Maria", "apellido": "Gomez"}  // Expresion de objeto literal
persona = {nombre: "Maria", apellido: "Gomez"}      // Forma abreviada de la expresion de objeto literal
```

Las claves en los objetos deben ser cadenas pero cuando las cadenas podrian ser identificadores (comenzar con letra, sin espacios) se puede omitir las comillas. Esta última es la forma habitual de acceder a los elementos de un objeto.

```
let persona = {nombre: "Juan", edad: 25, edad: 25}

// Acceso a una propiedad
persona["nombre"]  //> "Juan"  | Acceso a la propiedad nombre  | con corchetes
persona.nombre      //> "Juan"  | Acceso a la propiedad nombre | con punto
```

```
// Modificación de una propiedad
persona["nombre"] = "Pedro"  // Modificación de la propiedad nombre | con
corchetes
persona.nombre = "Pedro"     // Modificación de la propiedad nombre | con
punto
persona                     //> {nombre: "Pedro", edad: 25}

// Creación de una propiedad
persona["apellido"] = "Perez"  // Creación de la propiedad apellido | con
corchetes
persona.apellido = "Perez"     // Creación de la propiedad apellido | con
punto
persona                        //> {nombre: "Pedro", edad: 25, apellido:
"Perez"}

// Eliminación de una propiedad
delete persona["edad"]  // Eliminación de la propiedad edad | con corchetes
delete persona.edad     // Eliminación de la propiedad edad | con punto
persona  //> {nombre: "Pedro", apellido: "Perez"}

// Verificación de la existencia de una propiedad
"nombre" in persona  //> true  | Verificación de la existencia de la
propiedad nombre

// Objetos anidados
persona = {
    nombre: "Juan",
    edad: 25,
    direccion: {
        calle: "Av. Siempre Viva",
        número: 123
    }
}

persona.direccion //> {calle: "Av. Siempre Viva", número: 123}
persona.direccion.calle  //> "Av. Siempre Viva"  | Acceso a la propiedad
calle de la propiedad direccion
persona["direccion"]["calle"]  //> "Av. Siempre Viva"  | Acceso a la
propiedad calle de la propiedad direccion

persona.direccion.ciudad //> undefined | Cuando accedo a una propiedad que
no existe, devuelve undefined
```

Se puede consultar las claves y valores mediante funciones de libreria.

```
let persona = {nombre: "Juan", edad: 25, profesion: "Ingeniero"}
```

```
// Extraer las claves, valores y pares de un objeto
Object.keys(persona)    // ["nombre", "edad", "profesion"]
Object.values(persona)  // ["Juan", 25, "Ingeniero"]

Object.entries(persona) // [["nombre", "Juan"], ["edad", 25], ["profesion",
"Ingeniero"]]
```

En realidad todo en js son objetos aun los array y los datos primitivos aun estos tenga un trato especial.

Los array son objetos cuyas claves son numeros enteros consecutivos.

```
let numeros = [10, 20, 30]
Object.keys(numeros)    //> ["0", "1", "2"]
Object.values(numeros)  //> [10, 20, 30]

numeros.tipos = "decenas"
Object.keys(numeros)    //> ["0", "1", "2", "tipos"]
Object.values(numeros)  //> [10, 20, 30, "decenas"]
numeros                 //> [10, 20, 30, tipos: "decenas"]

numeros == {"0": 10, "1": 20, "2": 30, length: 3}
```

Los valores primitivos son objetos que se tratan como "invariantes" es decir que no se puede modificar.

```
let cadena = "Hola"       // Equivale a un array (que equivale a un objeto)

cadena[2]             //> "l"
cadena[2] = "X"       //> "X"
cadena                //> "Hola" | La cadena no se ha modificado

cadena.length = 2     //> 2
cadena                //> "Hola" | La cadena no se ha modificado

Object.keys(cadena)   //> ["0", "1", "2", "3"]
Object.values(cadena) //> ["H", "o", "l", "a"]
```

Ademas existen numeros objetos que forma parte de js como librerias. Por ejemplo las fechas.

```
// Las fechas son objetos
let d = new Date()
let n = d.getFullYear()
```

```
let e = new Date(2024,5,17) // 17 de junio de 2024
let f = e.getMilliseconds // milisegundos desde 1970

// Incluso los navegadores son objetos js
document.getElementById("year").innerHTML = "Año: " + n;
document.getElementById("fecha").innerHTML = "Fecha: " + e;
```

JSON

JSON (JavaScript Object Notation - Notación de Objetos de JavaScript) es una forma de intercambiar archivos. Consiste en un archivo de texto que contiene un objeto válido en JavaScript.

Este formato es fácil de leer para las personas y eficiente de procesar para las computadoras por lo que hecho muy popular en la actualidad. Si bien es un simple objeto de javascript el mismo debe ser escrito en forma completa, sin usar las formas abreviadas, es decir la clave de los objetos deben ser puesta entre comillas y solo se permite comillas dobles.

Existen funciones para convertir objetos en texto (para ser almacenado en un archivo de intercambio) y para convertir texto a objeto (para recuperar los valores almacenado)

En la actualizar todos los lenguajes de programación pueden leer y escribir archivo de JSON esto hacer muy popular a esta forma de compartir datos.

```
let productos = [
        {nombre: 'Cola Cola', precio: 100},
        {nombre: 'Pepsi Cola', precio: 80}
]

let productosJSON = JSON.stringify(productos) // convierte objeto en string
console.log(productosJSON)
//> '[{"nombre":"Cola Cola","precio":100},{"nombre":"Pepsi
Cola","precio":80}]'

let texto = '[{"nombre":"Cola Cola","precio":100},{"nombre":"Pepsi
Cola","precio":80}]'

let productosParseados = JSON.parse(texto) // convierte string en objeto
console.log(productosParseados)
//> [{ nombre:'Cola Cola', precio: 100 }, { nombre: 'Pepsi Cola', precio: 80
}]

console.log(JSON.stringify(productos,null, 4)) //> 4 es la cantidad de
espacios que se le agregan a la indentación

/*
```

```
[
    {
        "nombre": "Cola Cola",
        "precio": 100
    },
    {
        "nombre": "Pepsi Cola",
        "precio": 80
    }
]
*/
```

Asignación por valor o referencia

Un concepto muy interesante en JS es que los tipos primitivos se asignan por valor mientras que los tipos compuestos se aceden por referencia.

Es decir, cuando asigno una variable con un tipo primitivo, digamos un número, se realiza una copia, pero cuando asigno un tipo compuesto, digamos un array, no se copian los datos sino una referencia a los mismos.

Esto puede traer múltiples confusiones porque podemos alterar los datos sin ser nuestra intención pero es útil para manejar con mucha eficiencia el acceso y la modificación de los valores de un objeto.

```
// Con arrays
let a = [10, 20, 30]
b = a

b[0] = 100
b       //> [100, 20, 30]
a       //> [100, 20, 30]

// Con objetos
let o = {nombre: 'Juan', edad: 30}
let m = o
m.nombre = "Maria"
m       //> {nombre: "Maria", edad: 30}
o       //> {nombre: "Maria", edad: 30}

function ponerMayusculas(persona){
    persona.nombre = persona.nombre.toUpperCase()
}

o //> {nombre: "Maria", edad: 30}
ponerMayusculas(o)          // El paso de parámetros es por referencia
o //> {nombre: "MARIA", edad: 30}
```

```
function ponerMayusculasCadena(cadena){
    cadena = cadena.toUpperCase()
}

let nombre = "Juan"
ponerMayusculasCadena(nombre) // El paso de parámetros es por valor
nombre //> "Juan"

// Una curiosidad ...

const pares = [2, 4, 6, 8]                          // La referencia es
constante
pares[0] = 100 //> funcióna!!
pares // [100, 4, 6, 8]

pares = [4, 6, 8]  // No funcióna. La referencia es constante

Object.freeze(pares)    // Lo hace realmente constante
pares[0] = 2 //> No funcióna!!
pares // [100 4, 6, 8]
```

Cuando no se quiere este comportamiento se deben copiar los datos antes de modificarlos.

```
// Con los arrays
let original = [10, 20, 30, 40, 50];

let copia = []  // Forma tradicional usando for(;;)
for(let i = 0; i < original.length; i++){
    copia.push(original[i])
}

// Forma abreviada usando for(of)
copia = []
for(let x of original)
    copia.push(x);

copia[0] = 1000
copia           //> [1000, 20, 30, 40, 50]
original        //> [10, 20, 30, 40, 50]

// Hack para copiar arrays
copia = original.slice() //> [10, 20, 30, 40, 50]
copia = Object.assign([], original) //> [10, 20, 30, 40, 50]
```

Con los objetos es similar solo que debe copiarse clave por clave

```
let original = {nombre: "Juan", apellido: "Perez", edad: 20}
let copia = original
copia.nombre = "Pedro"
copia       //> {nombre: "Pedro", apellido: "Perez", edad: 20}
original    //> {nombre: "Pedro", apellido: "Perez", edad: 20}

// Copiar de objeto

let claves = Object.keys(original)

let copia = {}  // Forma tradicional con for(;;)
for (let i = 0; i < claves.length; i++) {
    let clave = claves[i]
    copia[clave] = original[clave]
}

copia = {}      // Forma más sencilla con for(of)
for (let clave of claves) {
    copia[clave] = original[clave]
}

// Forma más sencilla aun con con for(in)
copia = {}
for(let clave in original) {
    copia[clave] = original[clave]
}

// Hacks para copiar objetos

copia = Object.assign({}, original) // Forma más sencilla con Object.assign
copia = JSON.parse(JSON.stringify(original)) // Forma más sencilla con JSON
```

Generalización de Asignación

El operador "=" es mas completo de lo que parece. En especial cuando se usan en las declaraciones de variables.

Se puede usar en cascada, dentro de expresiones, se puede abreviar e incluso se puede usar para hacer asignaciones simultáneas.

```
let nombre="Juan", apellido="Perez", edad=30; // Declaramos y asignamos tres
variables
let a = b = c = 10; // Asignamos el valor 10 a las tres variables

// Asignación abreviada

a = a + 10  // a = 20  | Acumulador
```

```
a += 10      // a = 30  | Acumulador abreviado

a = a - 10   // a = 20  | Decremento
a -= 10      // a = 10  | Decremento abreviado

a = a * 10   // a = 100 | Multiplicador
a *= 10      // a = 100 | Multiplicador abreviado

//En general se puede hacer con cualquier operador aritmético
a %= 10      // a = 0   | Módulo abreviado
a &&= 10     // a = 0   | AND abreviado

a += 1       // a = 1   | Incremento
a++          // a = 2   | Incremento abreviado

a -= 1       // a = 1   | Decremento
a--          // a = 0   | Decremento abreviado
```

Un caso especial es cuando se quiere extraer valores de un array o un objeto se puede usar la desestructuración.

Estructurar un objeto es formarlo a través del uso de variables, desestructuralo es la operación opuesta, a partir de un objeto extraer a variables sus valores.

```
// Desectructuración de objetos

let nombre = "Juan", apellido = "Perez", edad = 30;

let persona = {nombre: nombre, apellido: apellido, edad: edad} //Estructuro
una persona a partir de las variables
persona = {nombre, apellido, edad} // Cuando la clave y la variable tengan
el mismo nombre, se puede simplificar

({nombre, apellido}) = {nombre: "Jose", apellido: "Gomez", edad: 25}
//Desestructuro un objeto
// Los () son necesarios porque sino el compilador interpreta que es un
bloque de código

let {nombre: n, apellido: a} = persona //Desestructuro un objeto y cambio el
nombre de las variables
// n = persona.nombre               // Declara y asigna un valor a "n"
// a = persona.apellido             // Declara y asigna un valor a "a"

// Desestructuración de arrays
let primero = 10, segundo = 20;
let numeros = [primero, segundo]; //Estructuro un array a partir de las
variables
[primero, segundo] = [100, 200]  //Desestructuro un array
```

```
// primero = 100
// segundo = 200

let [uno, dos, tres] = [1, 2, 3] //Desestructuro un array (creo y asigno valores a las variables)
// uno = 1
// dos = 2
// tres = 3

let [x, ,  ... resto] = [1, 2, 3, 4, 5] //Desestructuro un array y asigno el resto a una variable
// x = 1
// el segundo elemento se omite (observe la coma)
// resto = [3, 4, 5]
```

Operador "..."

Existe un operador especial para tratar con los objetos y array. El operador "...", expandir o spread nos permite manipular objetos.

Se puede pensar que el operador ... es como eliminar los "[]" en los array o los "{}" en los objetos

```
// Operador  ...  o expandir o spread

let numeros = [1, 2, 3, 4, 5];
let otros = [ ... numeros]   // Saca los [] y lo convierte en una lista de elementos | Copia array

otros[0] = 100
otros //> [100, 2, 3, 4, 5]
numeros //> [1, 2, 3, 4, 5]

otro = [ ... numeros, 1000]       // Agrego un elemento al final
otro = [1000,  ... numeros]       // Agrego un elemento al principio
otro = [1000,  ... numeros, 2000] // Agrego un elemento al principio y al final
otro = [ ... numeros,  ... numeros]  // Concatenar dos arrays (duplicar numeros)

// Con objetos también se puede hacer

let persona = {nombre: 'Juan', edad: 25}
let otra = {  ... persona } // Copia de un objeto

otra = {  ... persona, apellido: "Dias" } // Agregar un atributo al objeto
// equivale a { nombre: 'Juan', edad: 25, apellido: 'Dias' }
```

```
otra = {  ... persona, edad: 26 } // Modificar un atributo del objeto
// equivale a { nombre: 'Juan', edad: 25, edad: 26 } | Queda el ultimo valor

otra = {apellido: 'Juan',  ... persona} // Asumiendo que apellido es un
atributo del objeto persona

// Se puede usar para pasar argumentos a una función
```

Control de flujo

La programación propiamente dicha consiste en la toma de decisiones.

Mediante la toma de decisiones se eligen diferentes caminos de ejecución.
Esto puede usarse para elegir que se ejecuta o si se ejecuta reiteradas veces.

IF: Ejecución condicional

La ejecución condicional es el primer mecanismo para la toma de decisión.

La sentencia IF es la forma de materializar la ejecución condicional.

```
let edad = 20;

if(edad ≥ 18) // La forma mas básica de un if
    console.log('Eres mayor de edad');
// el 'else' es opcional.

let a = 10, b = 20
let m
if(a > b) // La forma mas básica de un if-else
    m = a
else
    m = b

if(a > b) { m = a } else { m = b } // En una sola linea.

if(a > b){ // Si hay mas de una linea de codigo en el bloque del if o else,
se usan llaves.
    m = a
    console.log('a es mayor que b')
}

// Los if pueden ir anidados
let c = 15  // encontrar el mayor de tres numeros a,b,c

if(a > b){
    if(a > c)   // if dentro de otro if
        m = a
```

```
    else
        m = c
} else {
    if(b > c)
        m = b
    else
        m = c
}

// O pueden ir en cascada
let nota
if(nota ≥ 90)
    console.log('A')
else if(nota ≥ 80)
    console.log('B')
else if(nota ≥ 70)
    console.log('C')
else if(nota ≥ 60)
    console.log('D')
else
    console.log('F')

// o en secuencia
let d = 15, e = 40
m = a // El mayor de 5 numeros a,b,c,d,e
if(b > m) m = b
if(c > m) m = c
if(d > m) m = d
if(e > m) m = e

// Dos usos especiales.
if(a > b)
    m = a
else
    m = b
// Cuando se usa el if para asignar un valor a una variable, se puede usar
el operador ternario
m = (a > b) ? a : b

// Otro uso aprovechando el cortocircuito

(a > b) && console.log('a es mayor que b') // Si la condicion es verdadera,
se ejecuta la segunda parte
// es equivalente a
if(a > b) console.log('a es mayor que b')
```

También existe la sentencia switch para reemplazar los if en cascada

```
switch(nota) {
    case 1:
        console.log("Excelente");
        break;
    case 2:
        console.log("Muy bien");
        break;
    case 3:
        console.log("Bien");
        break;
    case 4:
        console.log("Regular");
        break;
    case 5:
        console.log("Mal");
        break;
    default:
        console.log("Error");
        break;
}
```

Repetición

Otra esctrutura fundamental es la repetición del código.
Existen tres estructura que nos permite repetir la ejecución: while, do while y for.

while

Ejecuta un código mientras que se cumpla una condición. Cuando la condición es falta se termina la ejecución.
También puede salirse de un while usando un break (o un return)

Como la condición se verifica antes de la ejecución puede no ejecutarse nunca.

```
let i = 0;
while(i < 5){ // Imprime los numeros del 0 al 4
    console.log(i);
    i++;
}

i = 0
while(true){            // Bucle infinito
    console.log(i);
    i++;
    if(i == 5) break;   // Rompe el bucle
}
```

```
// Recorrer un array
let numeros = [1, 2, 3, 4, 5];
i = 0;
while(i < numeros.length){  // Imprime los numeros del array
    console.log(numeros[i]);
    i++;
}
```

Existe otra forma donde siempre se ejecuta al menos una vez ya que la condición se comprueba al final.

```
let i = 0
do {
    console.log(i)
    i++
} while (i < 10) // Siempre se ejecuta una vez.
```

for

Otra forma de realizar repeticiones es usando la sentencia for.
Esta es una forma compacta de hacer un while y consta de tres partes.
Una inicialización, una condición que comprueba antes, y expresión para ir al siguiente elemento.

```
let numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

let suma = 0;

// Sumar los elementos de un array
suma = 0

let i = 0                       // for(let i =0;  ...  ,  ...  )
while(i < numeros.length){      // for( ...  ; i < numeros.length ;  ... )
    let x = numeros[i]
    suma += x
    i++                         // for(  ...  ;  ...  ; i++)
}

suma = 0        // Lo mismo pero con un for
for(let i = 0; i < numeros.length; i++){
    let x = numeros[i]
    suma += x
}
```

```
// Listar los numeros pares del 10 al 20
for(let i = 10; i ≤ 20; i += 2){   // Comienza en 10, termina en 20, y se incrementa de 2 en 2
    console.log(i)
}

// for(of) Forma compacta de recorrer un array con un for
suma = 0
for(let x of numeros){
    suma += x
}

// for(in) Forma compacta de recorrer un objeto
let persona = { nombre: "Juan", edad: 25, sexo: "M" }
let claves = Object.keys(persona)   // Devuelve un array con las claves del objeto

for(let i = 0; i < claves.length; i++){ // Recorre el array de claves con un for tradicional
    let clave = claves[i]
    let valor = persona[clave]
    console.log(`${clave}: ${valor}`)
}

for(let clave of claves){       // Recorre el array de claves con un for(of) sobre las claves
    let valor = persona[clave]
    console.log(`${clave}: ${valor}`)
}

for(let clave in persona){      // Recorre las claves del objeto con un for(in)
    let valor = persona[clave]
    console.log(`${clave}: ${valor}`)
}

// Parentesis balancedados en una expresion
let expresion = "((a+b)*(c-d))"
let parentesis = 0
for(let letra of expresion){
    if(letra == "(") parentesis++;
    if(letra == ")") parentesis--;
    if(parentesis < 0) break;           // break sale del bucle
}
if(parentesis == 0)
    console.log("Parentesis balanceados");
```

Existe tres formas del for.

- La tradicional *for(inicial; condición; avance)* es la mas general.
- *for(of)* es una forma especializadas en recorrer una secuencia.
- *for(in)* es una forma especializada para recorrer las claves de un objeto.

En la forma tradicional todas las partes son opciones: *for(;;)* es un bucle infinito.

funciones

Las funciones son una forma básica de organizar el código. Tiene dos funciones: Evitar las repeticiones de código y hacer mas fácil de entender un programa.

Consiste en mini programas a los que se le asigna un nombre para poder invocarlos a voluntad.

las funciones tiene 4 partes.

- Un nombre para referenciar al código
- Una lista de parámetros para controlar el comportamiento de la función.
- El código propiamente dicho
- El retorno de un valor.

La declaración de una función solo la define. Para ejecutarla debemos invocarla usando paréntesis al final de la misma.

```
function menor(a,b){
    let m   // Variable local | Almacena el resultado
    if(a<b){
        m = a
    } else {
        m = b
    }
    return m    // Devuelve el resultado
}

menor(10,20) //> 10
menor(20,10) //> 10

// El return se puede usar para devolver un valor de una función, y también
para terminar la ejecución de una función.

function menor(a,b){ // Forma alternativa | Termina ni bien sepa el
resultado
    if(a<b) return a
    return b
}
```

Las funciones en JS son muy flexible con respecto a como se le pasa parámetros.
Los parámetros son variables locales se que asignan al invocar la función.
Lo particular de JS es que pueden se invocadas con mas valores o menos valores que los declarados.

Esto es normal y útil pero debe ser considerado especialmente.

```
function sumar(a,b){ return a+b }

sumar(1,2)      //> 3
sumar(1,2,3,4)  //> 3  (ignora los argumentos adicionales)
sumar(1)        //> NaN (porque b es undefined)
sumar()         //> NaN (porque a y b son undefined)

// Definir valores por defecto
function sumar1(a, b){          // Si no se pasa son undefined
    if(a===undefined) a=0       // Si es undefined se asigna 0
    if(b===undefined) b=0       // Si es undefined se asigna 0
    return a+b
}

function sumar2(a=0, b=0){      // Se puede asignar el valor por defecto
    return a+b
}

sumar1(1)       //> 1
sumar2(1)       //> 1
```

Los parámetros pueden ser definidos desestructurados

```
let contacto = {nombre: "Juan", apellido: "Perez"}

function nombreCompleto(persona) {
    return `${persona.nombre} ${persona.apellido}`
}

nombreCompleto(contacto); // "Juan Perez"

function nombreCompleto(persona) {
    let {nombre, apellido} = persona // Desestruturando variables locales
    return `${nombre} ${apellido}`
}

function nombreCompleto({ nombre, apellido }) {  // Desestructurando
parametros
    return `${nombre} ${apellido}`
}
```

```
nombreCompleto(contacto); // "Juan Perez"
```

Las funciones pueden ser "anidadas", es decir se puede definir una función dentro otra que no puede ser usada desde afuera.

```
let productos = [
    { id: 1, nombre: "Coca Cola", precio: 100 },
    { id: 2, nombre: "Pepsi", precio: 100 },
    { id: 3, nombre: "Fanta", precio: 100 },
    { id: 4, nombre: "Sprite", precio: 100 },
    { id: 5, nombre: "7up", precio: 100 },
    { id: 6, nombre: "Mirinda", precio: 100 },
]

function mostrarProductos(productos) {
    function mostrarProducto(producto){
        return `
                <div class="producto">
                        <h3>${producto.nombre}</h3>
                        <p>Precio: $${producto.precio}</p>
                        <button
onclick="agregarProducto(${producto.id})">Agregar</button>
                </div>
        `
    }

    let html = ""
    for(let producto of productos){
        html += mostrarProducto(producto)
    }

    return html
}
```

Cuando una función solo depende de los valores de entrada y la única salida es el valor que retorna se dice que es una 'función pura'. Para esto no debe usar variables externas ni producir efectos secundarios como almacenar un dato o mostrar algo en pantalla.

```
let precioMaximo = 100

function enPresupuesto(precio) {  // No es pura, depende de un valor externo
  return precio ≤ precioMaximo
}
```

```
function enPresupuesto(precio, precioMaximo) {  // No es pura, sale un valor por pantalla
    if(precio ≤ precioMaximo) {
       console.log("El precio supera el maximo permitido")
    }
}
```

Las funciones pueden retornar cualquier tipo de valor. Cuando se usan para crear un objeto se llama constructor.

```
function pares(maximo){
    let resultado = [];
    for (var i = 0; i ≤ maximo; i++){
        if (i % 2 == 0){
            resultado.push(i);
        }
    }
    return resultado;
}

console.log(pares(10)); // [0, 2, 4, 6, 8, 10]

function Persona(nombre, apellido, edad=18){
    return {nombre, apellido, edad};
}

let persona = Persona('Juan', 'Perez');
let {nombre, edad} = Persona('Juan', 'Perez', 25);
```

Las funciones pueden ser llamadas a si misma (recursividad)

```
function factorial(n) {
    if (n === 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

// Ordenar un array de numeros mediante el método de merge sort
function ordenar(lista) {
    function combinar(a,b){
        let result = []
        while(a.length > 0 && b.length > 0){// Mientras haya elementos
            if(a[0] < b[0]){                // Comparar el primer elemento d
```

```
                result.push(a.shift())      // Saca menor,a grega al
resultado
            }else{
                result.push(b.shift())
            }
        }
        return result.concat(a).concat(b);  // Agregar el resto de la lista
    }
    if (lista.length > 1) {
        let medio = Math.floor((inicio + fin) / 2)          // Dividir en dos
        let menor = ordenar(lista.slice(inicio, medio + 1)) // Ordena
primera
        let mayor = ordenar(lista.slice(medio + 1, fin + 1))// Ordenar
segunda
        return combinar(menor, mayor);      // Combinar las mitades en orden
    }
    return lista;
}
```

Las funciones en JS son ciudadanos de primera clase, es decir puede ser usada como cualquier otro tipo de JS. Pueden ser guardadas en variables, pasadas como parámetros o retornada como resultado.

La declaración de una función en realidad define una variable que referencia a un trozo de código.
Si se usan paréntesis al final se invoca o llama a la función. Si no se incluye los parámetros solo se obtiene una referencia a la función que puede ser almacenada en una variable.

```
function sumar(a,b){ // función declarada como sentencia
    return a+b;
}

let otra = sumar // Las funciones son objetos, por lo que se pueden asignar
a variables

sumar(10,20)     //> 30
otra(10,20)      //> 30 | Llama a la función sumar (otra tiene una referencia
a sumar)


let sumar1 = function(a,b){ // función declarada como expresion
    return a+b;
}

// Sumar y sumar1 son exactamente lo mismo
```

Un uso muy frecuente de las funciones como expresión es declarar funciones que se aplican a un objeto.
Cuando se tiene que definir una función para trabajar los datos de un objeto se puede incorporar la función misma dentro del objeto.

En este caso la función se usa como si fuera una propiedad a la que podemos invocar. No se puede usar desde afuera del objeto.

```
let persona = {
    nombre: "Juan",
    apellido: "Perez",

    nombreCompleto: function() {
        return `${persona.nombre} ${persona.apellido}`;
    }
}

console.log(persona.nombreCompleto()); //> Juan Perez

// El caso mas general es reemplazar el nombre del objeto por la palabra
reservada this

persona = {
    nombre: "Juan",
    apellido: "Perez",
    nombreCompleto: function() {
        return `${this.nombre} ${this.apellido}`;
    }
}
```

Otro uso de esta capacidad es la creación de librerías como Math o console.

```
const Math = {
    add: function(a, b) { return a + b},
    subtract: function(a, b) { return a - b},
    multiply: function(a, b) { return a * b},
    divide: function(a, b) { return a / b},
}

Math.add(2, 3)      //> 5
Math.subtract(5, 3) //> 2
Math.multiply(2, 3) //> 6


const l = console.log
l("Hola Mundo!") //> Hola Mundo!
```

```
// es equivalente a
console.log("Hola Mundo!") //> Hola Mundo!
```

También existe una forma moderna de declarar funciones de forma mas compacta. Se conoce como expresiones lambda o de la forma mas coloquial como "flecha gorda"

```
// Forma de definir una función en JS
function sumar1(a,b) { return a + b}

// funciones anónimas
const sumar2 = function (a,b) { return a + b } // se reemplaza 'function
sumar' por const 'sumar ='

// funciones flecha
const sumar3 = (a,b) => { return a + b } // Se reemplazar 'función ()' por
'() =>'

// Forma abreviada de la función anterior
const sumar4 = (a,b) => a + b // Cuando retorna un valor se puede omitir el
return y las llaves

const incrementar = a => a + 1 // Si solo tiene un parámetro se pueden
omitir los paréntesis

const duplicar = a => a * 2
const triplicar = a => a * 3
const esPar = a => a % 2 == 0
const esImpar = a => a % 2 != 0

const esMenor = (a,b) => a < b
const esMayor = (a,b) => a > b
```

Cuando se pasa una función como argumento o se retorna una función se llama funciones de orden superior (funciones que trabajan con funciones)

Esta forma de trabajar permite escribir código funcional (orientado a funciones) que es muy eficiente y elegante

Así como al pasar un parámetro podemos controlar el comportamiento de una función al pasar una función permite un mayor control de la función

```
// función que transforma un array

let numeros = [1, 2, 3, 4, 5];

// Crear un array con los dobles de los numeros
```

```
let dobles = []
for(let x of numeros){
    let y = x*2
    dobles.push(y)
}

let triples = []
for(let x of numeros){
    let y = x*3
    triples.push(y)
}

// función que transforma un array
function mapear(lista, accion){
    let resultado = []
    for(let x of lista){
        let y = accion(x)
        resultado.push(y)
    }
    return resultado
}

let duplicar = x => x*2
let triplicar = x => x*3

doble = mapear(numeros, duplicar)       //> [2, 4, 6, 8, 10]
triple = mapear(numeros, triplicar)     //> [3, 6, 9, 12, 15]

doble = mapear(numeros, x => x*2)       // Sin usar una variable intermedia
'duplicar'
triple = mapear(numeros, x => x*3)

// JS tiene un metodo 'map' que hace lo mismo
doble  = numeros.map(x => x*2)
triple = numeros.map(x => x*3)

// Función que filtra un array
let pares = []
for(let x of numeros){
    if(x % 2 == 0){
        pares.push(x)
    }
}

let esPar = x => x%2 == 0
// Función genérica para filtrar una lista que cumpla una condición
function filtrar(lista, condicion){
    let resultado = []
    for(let x of lista){
        if(condicion(x)){
```

```
            resultado.push(x)
        }
    }
    return resultado
}

pares   = filtrar(numeros, esPar)          //> [2, 4]
impares = filtrar(numeros, x => x % 2 != 0)  //> [1, 3, 5]

// JS tiene un método 'filter' que hace lo mismo
pares   = numeros.filter(esPar)
impares = numeros.filter(esImpar)

// Defino una función para sumar una lista
function sumar(lista){
    let resultado = 0
    for(let x of lista){
        resultado = resultado + x
    }
    return resultado
}

let suma = sumar(numeros)             //> 15

// Defino una función para multiplicar una lista
function multiplicar(lista){
    let resultado = 1
    for(let x of lista){
        resultado = resultado * x
    }
    return resultado
}

let producto = multiplicar(numeros)  //> 120

// Función genérica para reducir un array a un solo valor
// (Generalización de las dos funciones anteriores)
function reducir(lista, accion, valorInicial){
    let resultado = valorInicial
    for(let x of lista){
        resultado = accion(resultado, x)
    }
    return resultado
}

suma     = reducir(numeros, (a, b) => a + b, 0)  //> 15
producto = reducir(numeros, (a, b) => a * b, 1)  //> 120

// JS tiene un metodo 'reduce' que hace lo mismo
```

```
suma     = numeros.reduce((a, b) => a+b, 0)
producto = numeros.reduce((a, b) => a*b, 1)
```

Las funciones tiene por último una característica especial es que preserva las variables locales cuando son retornadas en una función. Se llama clausura y se usa intensamente

La clausura se produce cuando una variable local debe ser usada fuera de la función que la creo, esto se da cuando la función retorna otra función que usa dicho valor.
En ese caso JS no destruye la variable local sino que la preserva para ser usada posteriormente.

```
function contador(inicial=0){
    let cont = inicial
    return function() { return cont++ } // Retorna una función anónima
}

let a = contador(10);
let b = contador(20);
a() //> 10
a() //> 11
b() //> 20
b() //> 21
a() //> 12

// Función típica de reactjs
function useState(incial = 0){
    let estado = incial
    const get = () => estado
    const set = (nuevo) {
        console.log(`Cambiando estado de ${estado} a ${nuevo}`)
        estado = nuevo
    }
    return [get, set]
}

// React.js emplea una función useState para manejar el estado de los
componentes
let [getEstado, setEstado] = useState(10)
getEstado() //>  10
setEstado(20)   //> Cambiando estado de 10 a 20
getEstado() //> 20

setEstado(30)   //> Cambiando estado de 20 a 30
getEstado() //> 30

setEstado( getEstado() + 1 ) //> Cambiando estado de 30 a 31
```

Clases

JavaScript nos permite definir nuestras propias clases. Las clases permiten crear objetos. Las clases nos permite los datos (propiedades) con las funciones (métodos) para consultar o modificar el estado interno

JavaScript tiene muchas alternativas para crear clases.

Sin incorporar ningún concepto nuevo podríamos construir clases simplemente con una función que retorne un objeto.
En este caso cada vez que la función es llamada va a retornar una nueva instancia del objeto.

```
function Persona(nombre, edad){
    return {
        nombre: nombre,
        edad: edad,
        mostrar: function(){
            console.log(`${this.nombre} tiene ${this.edad} años`)
        }
    }
}

let juan = Persona("Juan", 30)
juan.mostrar()  //> "Juan tiene 30 años"

let maria = Persona("Maria", 20)
maria.mostrar() //> "Maria tiene 20 años"
```

Existe una forma especial de usar las funciones que están diseñada especialmente para crear objetos. En este caso se dice que se usa la función como constructor.
Es equivalente a la función anterior excepto que no hace falta crear y retornar los objetos. Basta usar "this." para referenciar las campos que van a tener el objeto. En la invocación se debe anteponer la palabra new.

```
function Persona(nombre, edad){
    this.nombre = nombre;
    this.edad = edad;
    this.mostrar = function(){
            console.log(`${this.nombre} tiene ${this.edad} años`)
    }
}

let juan = new Persona("Juan", 30)
juan.mostrar()  //> "Juan tiene 30 años"
```

```
    let maria = new Persona("Maria", 20)
    maria.mostrar() //> "Maria tiene 20 años"
```

A partir de la version ES5 existe una nueva sintaxis para hacer exactamente lo mismo que al funciones constructoras.

```
class Persona {
    constructor(nombre, edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    mostrar(){
        console.log(`${this.nombre} tiene ${this.edad} años`)
    }
}

const juan = new Persona('Juan', 30);
const maria = new Persona('Ana', 20);

juan.mostrar(); //> "Juan tiene 30"
maria.mostrar(); //> "Maria tiene 20"
```

Las clases, ademas de ser mas faciles de escribir permite crear nuevas clases que hereden (copian) la caracteriticas

Este mecanismo permite especilizar una clase agregandole comportamientos nuevos y retulizando todos los existenes.

```
class Empleado extends Persona {    // Herencia de la clase Persona (copia
las propiedades y métodos de la clase Persona)
    constructor(nombre, edad, salario) {
        super(nombre, edad);
        this.salario = salario;
    }

    mostrar() {
        console.log(`${this.nombre} tiene ${this.edad} años y gana
$${this.salario}`)
    }

    aumentar(monto) {
        this.sueldo += monto
    }
```

```
}

const pedro = new Empleado('Pedro', 27, 1500);
const luisa = new Empleado('Luisa', 35, 2000);

pedro.mostrar(); //> //Pedro tiene 27 años y gana $1500"

pedro.aumentar(100)
pedro.mostrar(); //> //Pedro tiene 27 años y gana $1600"


luis.mostrar(); //> Luisa tiene 35 años y gana $2000
```

Por ultimo las clases permite la creación de propiedades con acceso controlado. Normalmente una propiedad es un campo de dato, pero cuando esos campos requiere controlar como se accede a ellos se pueden usar propiedades calculadas

```
    class Producto {
        constructor(nombre, precio, cantidad){
            this.#nombre = nombre                    // #nombre es una
variable privada (solo valida dentro de la clase)
            this.#precio = precio
            this.#cantidad # cantidad
        }

        get nombre(){ return this.#nombre}      // Propiedad de solo lectura
        get precio(){ return this.#precio}
        set precio(nuevo){
            if(nuevo ≤ 0) return    // Rechaza los precios negativos
            this.#precio = nuevo
        }
        get cantidad(){ return this.#cantidad}
        set cantidad(nuevo){
            if(cantidad < 0) return  // Rechaza cantidades negativas
            this.#cantidad = nuevo
        }

        get importe() { this.precio * this.cantidad } // Propiedad
calculada.
   }

   let coca = new Producto("Cola Cola", 1400, 4)
   console.log(`Hay ${coca.cantidad} botellas de ${coca.nombre} a un precio
unitario de $${coca.precio} lo que hace un importe de ${coca.importe}`)
   //> "Hay 4 botellas Coca Cola a un precio unitario de $1400lo que hace un
importe de $5600"

    coca.precio = -10  // Operacion rechazada
```

```
    console.log(`El precio es de $${coca.precio}`);

    coca.precio = 1000
    console.log(`El precio es de $${coca.precio}`);
```

Con las clases de pueden representar estructura complejas que modelen entidades.
Por ejemplo una agenda contiene multiples contactos que a la vez tienen nombre, apellido, telefonos, domicilios y email.

Los domicilios, telefonos y email son direcciones: fisica, telefonica y electrónica.
En este caso la agenda permite buscar todos los contactos que contenga un texto dado.
Para ello le pregunta a cada contacto

si contiene ese texto, este a su vez de pregunta a cada una de las direcciones si contiene el texto.
Cada clase sabe como mostrase (toString) y como determinar si el texto pasado esta en alguna de sus propiedades (contiene)

```
    // Agenda es una clase que contiene una lista de contactos
    class Agenda {
        constructor() {
            this.contactos = [];
        }

        agregar(contacto){
            this.contactos.push(contacto);
        }

        buscar(nombre){ // Busca los contactos que contienen el texto
indicado
            return this.contactos.filter(c => c.contiene(nombre));
        }
    }

    // Contacto es una clase que representa una persona con nombre,
apellido, telefonos y domicilios
    class Contacto {
        constructor(nombre, apellido,  ... direcciones) {
            this.nombre = nombre;
            this.apellido = apellido;
            this.direcciones = direcciones || [];
        }

        get nombreCompleto() {
            return `${this.nombre} ${this.apellido}`;
        }
```

```
        iniciales () {
            return `${this.nombre[0]}${this.apellido[0]}`;
        }

                // Busca el texto en el contacto (nombre, apellido, o
direcciones)
        contiene(texto){
            return this.nombreCompleto.includes(texto)
                || this.email().includes(texto) || this.iniciales() ===
texto
                || this.direciones.some(t => t.contiene(texto))
        }

        agregar(direccion) {
            this.direcciones.push(direccion);
        }

        telefonos() {
            return this.direcciones.filter(d => d instanceof Telefono);
        }

        domicilios() {
            return this.direcciones.filter(d => d instanceof Domicilio);
        }

        toString() {
            return `${this.nombreCompleto}
\n${this.direcciones.join('\n')}`;
        }
    }

    // Direccion puede ser telefonos, domicilios o email
    class Direccion {
        constructor(tipo) { this.tipo = tipo }
        contiene(texto) { return this.tipo.includes(texto) }
    }

    // Teléfono es una direccion telefonica con numero y tipo
    class Telefono extends Direccion {
        constructor(numero, tipo = 'celular') {
            super(tipo)
            this.numero = numero;
        }

        contiene(texto){    // Busca el texto en el teléfono (numero o tipo)
                return super.includes(texto)
                       || this.numero.toString().includes(texto)
        }

        toString(){ return `Teléfono: ${this.numero} (${this.tipo})` }
```

```
    }

    // Domicilio es es una direccion fisica con calle, numero y localidad
    class Domicilio extends Direccion {
        constructor(calle, numero, localidad = 'Tucuman', tipo = 'casa') {
            super(tipo)
            this.calle = calle;
            this.numero = numero;
            this.localidad = localidad;
        }

                // Busca el texto en el domicilio (calle, numero o
localidad)
        contiene(texto){
            return super.contiene(texto)
                || this.calle.includes(texto)
                || this.numero.includes(texto)
                || this.localidad.includes(texto);
        }

        toString(){
         return `Domicilio:${this.calle} ${this.numero} (${this.localidad})`
        }
    }

    // Email es una direccion electronica
    class Email extends Direccion{
        constructor(direccion, tipo='personal') {
            super(tipo)
            this.direccion = direccion;
        }

                 // Busca el texto en el email (direccion)
        contiene(texto){
            return super.contiene(texto)
                    || this.direccion.includes(texto);
        }

        toString(){ return `Email: ${this.direccion} (${this.tipo})` }
    }

    let juan = new Contacto('Juan', 'Perez');

    juan.agregar(new Telefono('3811234567'));
    juan.agregar(new Telefono('3817654321', 'fijo'));

    juan.agregar(new Domicilio('Mendoza', 123));
    juan.agregar(new Domicilio('San Martin', 456, 'Yerba Buena'));

    juan.agregar(new Email('jperez@mail.com', 'trabajo'))
```

```
    console.log(juan);

    // Acceso a los atributos de un contacto
    juan.nombre = 'Juan Carlos';
    juan.direcciones.length //> 2

    juan.domicilios.length //> 2
    juan.domicilios[1].localidad = "Tafi Viejo" // Cambia la localidad del
domicilio 1 del contacto

    let maria = new Contacto('Maria', 'Lopez');
    maria.agregar(new Telefono('3819876543', 'fijo'));
    maria.agregar(new Domicilio('25 de Mayo', 123));
    maria.agregar(new Email('mlopez@mail.com', 'personal'))

    let agenda = new Agenda();
    agenda.agregar(juan);
    agenda.agregar(maria);
    agenda.agregar(new Contacto('Jorge', 'Lopez'));

    agenda.agregar(new Contacto('Carlos', 'Gomez',
        new Domicilio('Calle Falsa', 123, 'Springfield'),
        new Email('cgomez@mail.com'),
        new Telefono('3811234567'),
        new Telefono('3817654321', 'fijo')
    ))

    console.log(agenda);
    let j = agenda.buscar('Juan');
    console.log(j)

    console.log("> Contactos que contienen 'Lopez'")
    for(let contacto of agenda.buscar('Lopez')){
        console.log(`- ${contacto.nombreCompleto}`)
    }

    console.log("> Contactos que contienen '381'")
    for(let contacto of agenda.buscar('381')){
        console.log(`- ${contacto.nombreCompleto}`)
    }
```

Módulos

Los módulos son una forma de organizar y reutilizar código en JavaScript. Permiten dividir una aplicación en partes más pequeñas y separadas, lo que facilita el mantenimiento y la colaboración en proyectos grandes.

Los módulos son una forma estándar de definir y exportar código reutilizable en JavaScript. Cada módulo representa un archivo independiente con su propio ámbito y puede contener funciones, clases, variables y otros elementos.

Los módulos se importan y exportan entre sí para compartir funcionalidad.

```
// --- Archivo mi-modulo.js ---

// Exportar una constante
export const MI_CONSTANTE = 42;

// Exportar una función
export function miFuncion(){ ... }

// Función privada que no puede usar externamente
function otraFuncion(){ ... }

// export { miFuncion, MI_CONSTANTE } ⟵ Sintaxis alternativa
```

```
// --- Archivo probar.js ---
// Importar una función o constante
import { miFuncion, MI_CONSTANTE } from './mi-modulo.js';
```

Los elementos que pueden ser usado externamente debe ser declarados como exportable. Es decir que pueden ser usado desde otro archivo. Se puede declarar en forma individual poniendo la palabra clave 'export' delante de la función, variable, constante, etc que quiere ser usado externamente. También se puede exportar funcionalidad poniendo la palabra 'export' y una lista de todos los objetos exportables entre {}

Para usarse de hace lo opuesto, el archivo en donde se use el modulo debe declarar cuales son los elementos que se van usar. Va la palabra 'import' luego la lista de elementos a importar, luego 'from' y por último la ubicación del archivo donde esta el código a usar.

Los módulos pueden importar algunas funciones y exportar otra. Por ejemplo es normal en ReactJS definir un nuevo componente de la siguiente forma:

```
// Boton.js
import React from 'react';

// Definimos el componente Boton como una función
const Boton = ({ onClick, children }) ⇒ (
    <button onClick={onClick}>
      {children}
    </button>
```

```
)

export default Boton;
```

Luego se pueden usar el nuevo componente creado de la siguiente manera:

```
// Confirmar.js

import React from 'react';   // Importamos React
import Boton from './Boton'; // Importamos el componente Boton

const Confirmar = ({pregunta, alConfirmar}) => (
    <div>
      <p>{pregunta}</p>
      <Boton onClick={() => alConfirmar(true)}>Aceptar</Boton>
      <Boton onClick={() => alConfirmar(false)}>Cancelar</Boton>
    </div>
)
// Exportamos el componente Confirmar
export default Confirmar;
```

Asincrónico

Normalmente la ejecución de los programa se realiza en forma secuencial (o sincrónica) , línea por línea en el orden que esta escrito. Pero hay circunstancias que las tareas deben ser realizadas simultáneamente (o en paralelo o en forma asincrónica) de esta manera puede aprovechar los tiempos.

Callback

En el navegador los eventos del usuario (pulsa un botón, presionar una tecla, hacer clic)

```
function mostrarMensaje(){
        alert("Pulsaste el boton")
}

var boton = document.getElementById('miBoton');
boton.addEventListener('click', mostrarMensaje)
```

En este ejemplo la función "mostrarMensaje" se registra en el botón para ser llamada cuando el usuario hace un click. Las funciones que son pasadas para ser llamadas cuando se realice una operación asincrónica se llama callback y es la forma de saber que la tarea ha terminado.

Ejecución diferida

Existen dos funciones que permite agendar la ejecución de una función en el futuro.
No es parte del lenguaje, es una librería que esta presente tanto en el navegador como en nodejs

La función setTimeout recibe un callback que será ejecutado ni bien pase el tiempo especificado en milisegundos. setInterval realiza lo mismo pero repite la llamada cada cierto intervalo de tiempo

```
// Esta función muestra un texto y la hora en que se ejecuta
const log = texto ⇒
       console.log(texto.padEnd(30),new Date().toLocaleTimeString())

// Definimos dos funciones que simulan tareas
let  tarea1 = () ⇒ log("2. Ejecutando Tarea 1")
let  tarea2 = () ⇒ log("3. Ejecutando Tarea 2")

console.clear()
console.log(">> Ejecución de tareas sincrónicas <<\n")
log("1. Inicio (sincrónico)")
tarea1()
tarea2()
log("4. Fin")

//> Ejecución de tareas sincrónicas
//1. Inicio                      11:56:46
//2. Ejecutando Tarea 1          11:56:46
//3. Ejecutando Tarea 2          11:56:46
//4. Fin                         11:56:46

// Callbacks
console.log("\n>> Ejecución de tareas asíncronas <<\n")
log("1. Inicio (asíncrono) ")
setTimeout(tarea1, 2000)
setTimeout(tarea2, 1000)
log("4. Fin")

//> Ejecución de tareas asíncronas
//1. Inicio                      11:56:46
//4. Fin                         11:56:46
//3. Ejecutando Tarea 2          11:56:47
//2. Ejecutando Tarea 1          11:56:48

console.log("\n>> Ejecución de tareas asíncronas repetidas <<\n")
log("1. Inicio (asíncrono periódicas)")
setInterval(tarea1, 1000)
log("3. Fin")
```

```
//> Ejecución de tareas asíncronas repetidas
//1. Inicio                       11:56:46
//3. Fin                          11:56:46
//2. Ejecutando Tarea 1           11:56:47
//2. Ejecutando Tarea 1           11:56:48
//2. Ejecutando Tarea 1           11:56:49
//2. Ejecutando Tarea 1           11:56:50
// Continúa hasta presionar CTRL + C
```

Promesas

Una tercera forma de ejecutar funciones asincrónica es usando promesas.
Las promesas permite iniciar una tarea en paralelo e informar cuando la operación se completo o en su defecto si la función falla.

Muchas funciones de librería que trabaja con operaciones que pueden ser lentas incorporan este comportamiento. La función fetch, por ejemplo, permite traer el contenido de un sitio. Como esta operación puede ser lenta, al invocarla inmediatamente retorna el control a la línea siguiente de la llamada, sin embargo la función se seguirá ejecutando en paralelo y avisará cuando sea resuelta o rechazada.

```
const url = "https://jsonplaceholder.typicode.com/posts/1"

// Callback para convertir a JSON
const convertirJSON = response => {
    log("2. Datos obtenidos")
    return response.json()  // Devuelve una promesa
}

// Callback para mostrar el JSON
const mostrarJSON = json => {
    log("3. Datos: " + JSON.stringify(json))
}

console.clear()
log("1. Antes del fetch")
fetch(url)
    .then(convertirJSON)
    .then(mostrarJSON)
    .catch(error => console.error("Error: ", error))

log("4. Después del fetch")

//> 1. Antes del fetch                                  02:21:37
//> 4. Después del fetch                                02:21:37
//> 2. Datos obtenidos                                  02:21:37
```

```
//> 3. Datos: {"userId":1,"id":1,"title":"sunt aut  ... 02:21:37
```

Una particularidad de las promesas es que la funciones que actúan como callback se deben llamar como .then() para el caso de ser exitosa o .catch() para el caso de los errores.

Si queremos crear nuestras propias promesas podemos usar la clase Promise y pasarle un callback que para las operaciones de resolver y reject

Al crearse la promesa el código que esta dentro del callback empieza a ejecutar pero lo hace en paralelo ya que el control es retornado inmediatamente para seguir ejecutando la línea siguiente.

```
let tareaLenta = new Promise((resolve, reject) => {
    // --- Ejectamos el código lento aquí ---
    const resultado = true // true para simular que la tarea fue exitosa
    if(resultado) {
        resolve("La tarea fue exitosa")
    } else {
        reject("La tarea falló")
    }
})
// Ya esta realizando la tarea
log("1. Antes de la tarea lenta")
tareaLenta
    .then(resultado => log("2. " + resultado))
    .catch(error => log("2. " + error))
log("3. Después de la tarea lenta")

//> 1. Antes de la tarea lenta                          02:21:37
//> 3. Después de la tarea lenta                        02:21:37
//> 2. La tarea fue exitosa                             02:21:47
```

La implementación de las promesas se suelen hacer usando try /catch. Esta estructura de control ejecuta un código dentro del try y si se produce algún error durante la ejecución continua ejecutando en el bloque catch

```
// Ejemplo de promesa con try catch
const otraTarea = new Promise((resolve, reject) => {
    try {
        // --- Ejectamos el código lento aquí ---
        resolve("Tarea exitosa")
    } catch (error) {
        reject(error)
    }
```

```
})

otraTarea
        .then( console.log )    // Usamos console.log para mostrar el exito
        .catch( console.error ) // Usamos console.error para mostrar el
error
```

- ## Async / await

En JavaScript, async y await son dos palabras clave que se utilizan para trabajar con promesas de una manera más cómoda y legible. La palabra clave async se utiliza para declarar una función asíncrona, que devuelve una promesa. La palabra clave await se utiliza dentro de las funciones async para esperar a que una promesa se resuelva antes de continuar con la ejecución del código.

```
async function traerDatos(){
    const resp = await fetch('https://jsonplaceholder.typicode.com/todos/1')
    const json = await resp.json()
    return json
}

log("1. Antes de traer datos")
traerDatos() // continua sin esperar
        .then( json => log("2. Datos obtenidos: ${json}") // Al finalizar
log("3. Después de traer datos")

//> 1. Antes de esperar             02:21:37
//> 3. Despues de traer datos       02:21:37
//> 2. Datos obtenidos: { ... }     02:21:47
```