

Javascript básico

Fundamentos

Todo lenguaje de programación está compuesto por dos elementos

- Datos
- Operaciones

Fundamentos

Todo lenguaje de programación está compuesto por dos elementos:

- Datos
 - Números
 - Textos
 - Fechas
 - ...
- Operaciones

Fundamentos

Todo lenguaje de programación está compuesto por dos elementos:

- Datos
 - Números
 - Textos
 - Fechas
 - ...
- Operaciones
 - Aritmética
 - Comparación
 - Concatenación
 - ...

Datos

Datos

Hay dos elementos principales en el ámbito de los datos

- Valores
- Variables

Valores

Javascript tiene múltiples **tipos** de valores

- Number
- String
- Boolean
- Undefined
- Object

Valores

Javascript tiene múltiples **tipos** de valores

- **Number**
 - 0
 - 27
 - 480.23
- String
- Boolean
- Undefined
- Object

Valores

Javascript tiene múltiples **tipos** de valores

- **Number**
 - 0
 - 27
 - 480.23 ← A los números decimales se les llama **float** en programación
- String
- Boolean
- Undefined
- Object

Valores

Javascript tiene múltiples **tipos** de valores

- Number
- **String** (cadenas de texto)
 - "Hello world"
 - "A"
 - ""
- Boolean
- Undefined
- Object

Valores

Javascript tiene múltiples **tipos** de valores

- Number
- **String** (cadenas de texto)
 - "Hello world"
 - "A" ← Pueden tener un solo carácter
 - "" ← Pueden estar vacías
- Boolean
- Undefined
- Object

Valores

Javascript tiene múltiples **tipos** de valores

- Number
- String
- **Boolean**
 - true
 - false
- Undefined
- Object

Valores

Javascript tiene múltiples **tipos** de valores

- Number
- String
- Boolean
- **Undefined** (valor vacío)
- Object

Variables

Variables

Son contenedores que almacenan datos

```
let alumno = "Alejandro"
```

Variables

Son contenedores que almacenan datos

```
let alumno = "Alejandro"
```

- **let** declara que lo que sigue es una variable
- **alumno** es el nombre que le hemos dado a esa variable
- **“Alejandro”** es el valor que se almacena en la variable **alumno**

Variables

Podemos crear múltiples variables

```
let alumno = "Alejandro"  
let edad = 29  
let aprobado = true  
let numeroTelefono = undefined
```

Cuando hay varias líneas, se ejecutan una detrás de otra

Variables

Podemos separar la **declaración** y la **igualación** de una variable

```
let alumno  
alumno = "Alejandro"
```

Variables

Podemos sobrescribir el valor que guarda una variable

```
let alumno  
alumno = "Alejandro"  
alumno = "Irene"
```

Imprimir en consola

Podemos imprimir valores y variables utilizando la función **console.log()**

```
console.log("Hello World")
```

- **console.log()** imprime el valor que recibe entre paréntesis ()

Demo

Cómo ejecutar Javascript

Ejercicio Introducción

- Crea una subcarpeta **workspace/javascript** donde pondrás todos los ejercicios de Javascript

Ejercicio variables

Crea tres variables que contengan los siguientes textos:

- Alfa
- Beta
- Gamma

Imprimelas en la consola de Chrome en el orden correcto

Variables

Hay tres tipos de variables

```
var x = 1  
let y = 1  
const z = 1
```

- **var** es la forma antigua de declarar variables, tiene algunos inconvenientes
- **let** es la forma moderna y correcta de declarar variables
- **const** se utiliza para declarar **constantes** (variables que no se pueden sobrescribir)

Ejercicio constantes

Crea una constante e intenta sobrescribir el valor

Operaciones

Aritmética

Podemos aplicar operaciones matemáticas a los valores de **tipo number**

```
let a = 1 + 1  
let b = 2 - 1
```

```
console.log(a) // ?  
console.log(b) // ?
```

Aritmética

```
let a = 1 + 1
```

- Qué es $1 + 1$?

Aritmética

```
let a = 1 + 1
```

- Qué es $1 + 1$?

$1 \rightarrow$ valor

$+$ \rightarrow **operador**

$1 \rightarrow$ valor

- A el conjunto lo llamamos una **expresión**
- El resultado de **evaluar** la **expresión** es un valor

Aritmética

También podemos sumar **variables** que contengan valores number

```
let a = 1 + 1  
let b = 2 - 1  
let c = a + b
```

```
console.log(c) // ?
```

Aritmética

Javascript dispone de varios operadores aritméticos:

```
let suma = 1 + 1      // 2
let resta = 2 - 1     // 1
let multi = 2 * 3     // 6
let division = 6 / 2  // 3
let residuo = 7 % 2   // 1
```

Podemos forzar el orden usando paréntesis:

```
let resultado = 2 + 2 * 2      // ?
let resultado = (2 + 2) * 2    // ?
let resultado = (2 + 2) / (1 + 1) // ?
```


Ejercicio aritmética

Calcula el resultado de esta operación y almacénalo en una variable

$$\frac{1}{2}(4 \cdot 5) - 2^3$$

Comparación

Podemos comparar valores usando:

```
// > significa mayor que  
let resultado = 1 > 2
```

- Cuál es el resultado ?
- De qué tipo es el resultado?

Comparación

Podemos comparar valores usando:

```
// > significa mayor que  
let resultado = 1 > 2
```

- Cuál es el resultado ? **false**
- De qué tipo es el resultado? **Boolean**

Comparación

Podemos comparar valores **y variables**:

```
let edad = 36  
let mayorDeEdad = edad > 18 // true
```

Comparación

Javascript dispone de varios operadores de comparación

```
2 > 2 // false
```

```
2 < 2 // false
```

```
2 >= 2 // true
```

```
2 <= 2 // true
```

```
2 == 2 // true
```

Podemos forzar el orden usando paréntesis

```
let resultado = 2 > (2 * 2) // true
```

Comparación

Podemos usar el operador de igualdad para **cualquier tipo** de valor

```
2 == 2           // true
"abc" == "Abc"   // false
"abc" == "cbd"   // false
```

Podemos forzar el orden usando paréntesis:

```
let resultado = 2 > (2 * 2)           // true
```

Para qué queremos estos Booleanos?

Condicionales

Condicionales

Podemos controlar el flujo de nuestro programa mediante **condicionales**

```
if(booleanValue) {  
    // Se ejecuta solo si el booleano es true  
} else {  
    // Se ejecuta solo si el booleano es false  
}
```

Condicionales

Podemos controlar el flujo de nuestro programa

```
let edad = 36
let mayorDeEdad = edad > 18

if(mayorDeEdad) {
  console.log("you can vote")
} else {
  console.log("you can't vote")
}

console.log("end")
```

Condicionales

El **if** también admite **expresiones**

```
let edad = 36

if(edad > 18) {
  console.log("you can vote")
} else {
  console.log("you can't vote")
}
```

Condicionales

El **else** es opcional

```
let edad = 36
```

```
if (edad > 18) {  
    console.log("you can vote")  
}
```

Ejercicio condicionales

- Crea una **variable** que contenga un número cualquiera
- A continuación utiliza un condicional para que
 - Si el número es **más grande de 10**, imprime la **mitad** de ese número
 - Si el número es **más pequeño o igual que 10**, imprime el **doble** de ese número
- Comprueba que funciona para ambos casos cambiando el número asignado a la variable

Ejercicio condicionales II

- Crea una **variable** que contenga un número cualquiera de 0 a 100
- A continuación utiliza un condicional para que
 - Si el número es **0** imprime: El motor está apagado
 - Si el número es **100** imprime: El motor está a máxima potencia
 - En cualquier otro caso:
 - imprime: El motor está a potencia media:
 - Imprime el número
- Comprueba que funciona para todos los casos

Ejercicio condicionales III

- Crea una **variable** llamada **activo** que sea **true** o **false**
- Crea otra **variable** que contenga un número 1 o 2
- A continuación utiliza un condicional para que
 - Si **activo** es **false** imprime una string: el programa no está activo
 - Si **activo** es **true**, entonces:
 - Si el número es **igual a 1**, imprime: Hello world
 - Si el número es **igual a 2**, imprime: Hello world dos veces
- Comprueba que funciona para los tres casos cambiando los valores de las variables

Truthiness

If no solo acepta valores booleanos

Todos los valores equivalen a **true** excepto:

1. `false`
2. `null`
3. `undefined`
4. `<=0`
5. `NaN`
6. `''`

Truthiness

```
let saldo = 2344
```

```
if(saldo) {  
  console.log("La cuenta tiene saldo")  
}
```

Ejercicio truthiness

- Determina si estas expresiones son **truthy** o **falsey**
 - **"Abc"**
 - **20**
 - **"20"**
 - **-20**
 - **"0"**
 - **0**
 - **" "**
 - **"undefined"**
 - **undefined**
 - **null**

Operadores de igualdad

Loose equality (==)

- Intenta **type coercion**

| | |
|---------------------------------|----------------------|
| <code>77 == "77"</code> | <code>// true</code> |
| <code>0 == false</code> | <code>// true</code> |
| <code>undefined == false</code> | <code>// true</code> |

Strict equality (===)

- Comprueba igualdad en **valor** y en **tipo**

| | |
|----------------------------------|-----------------------|
| <code>77 === "77"</code> | <code>// false</code> |
| <code>0 === false</code> | <code>// false</code> |
| <code>undefined === false</code> | <code>// false</code> |

Unary operators

Podemos combinar condiciones usando **unary operators**

```
if(a > 3 && b < 10){ // }      // AND - Si se cumplen ambas condiciones  
if(a < 3 || c == 0) { // }     // OR - Si se cumple una de las condiciones
```

Unary operators

```
if(a > 3 && b < 10){ // }      // AND - Si se cumplen ambas condiciones  
if(a < 3 || c === 0) { // }    // OR - Si se cumple una de las condiciones
```

Podemos forzar el orden con paréntesis

```
// Si se cumple las dos primeras condiciones o la segunda  
if((a > 3 && b < 10) || c === 0) { // }
```

Ejercicio condicionales III - Unary operators

- Refactoriza el código anterior (condicionales III) utilizando **unary operators**

Bucles

Bucles

Los bucles nos permiten ejecutar piezas de código **repetidamente** mientras se **cumplan determinadas condiciones**

Vamos a ver dos tipos

- Bucles while
- Bucles for

Bucles while

Ejecutan su **body** mientras se cumpla la condición entre paréntesis

```
let i = 0 // i de index (esto es un contador)

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

Bucles while

Ejecutan su **body** mientras se cumpla la condición entre paréntesis

```
let i = 0
```

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

```
// 0
```

```
// 1
```

```
// 2
```

```
// 3
```

```
// 4
```

Bucles while

Dos partes: **header** y **body**

```
let i = 0
```

```
while (i < 5) {
```

← Header

```
  console.log(i)
```

```
  i = i + 1
```

```
  ...
```

← Body

```
}
```

Bucles while

Qué hace este fragmento de código?

```
while(false) {  
    console.log("Hello world")  
}
```

Bucles while

Qué hace este fragmento de código?

```
while(true) {  
    console.log("Hello world")  
}
```

Bucles for

```
let i = 0
```

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

Los bucles for nos permiten expresar lo mismo de forma más compacta

```
for(let i = 0; i < 5; i = i + 1) {  
  console.log(i)  
}
```

Bucles for

```
for(let i = 0; i < 5; i = i + 1){  
  console.log(i)  
}
```

El **header** (parte superior) del bucle for tiene tres partes

```
for(declaración índice; condición; modificación índice){  
  // ...  
}
```

Bucles for

Podemos modificar cualquiera de las tres partes

```
for(declaración índice; condición; modificación índice){  
    // ...  
}
```

Qué imprime este bucle?

```
for(let i = 2; i <= 8; i = i + 2){  
    console.log(i)  
}
```


Bucles for

Podemos modificar cualquiera de las tres partes

```
for(declaración índice; condición; modificación índice){  
    // ...  
}
```

Qué imprime este bucle?

```
for(let i = 2; i <= 8; i = i + 2){  
    console.log(i)  
}
```

```
// 2
```

```
// 4
```

```
// 6
```

```
// 8
```

Ejercicio loop

- Imprime todos los números del 1 al 10 usando un **bucle for**

Ejercicio loop II

- Imprime todos los números impares del 0 al 10 usando un **bucle for**

Ejercicio loop III

- Imprime todos los números impares del 0 al 10 usando **este bucle for**:

```
for(let i = 0; i < 10; i = i + 1) {
```

```
}
```

Funciones

Funciones

Encapsulan fragmentos de código

```
function saluda() {  
    console.log("Hola")  
    console.log("Soy un programa")  
}
```

```
// De momento, no se imprime nada
```

Qué podemos **llamar** siempre que queramos

```
saluda()
```

```
// Hola
```

```
// Soy un programa
```

Funciones

Encapsulan fragmentos de código

```
function saluda() {  
  console.log("Hola")  
  console.log("Soy un programa")  
}
```

← Declaración de la función

Qué podemos **llamar** siempre que queramos

```
saluda()  
  
// Hola  
// Soy un programa
```

← Llamada a la función

Funciones

Encapsulan fragmentos de código

```
function saluda() {  
    console.log("Hola")  
    console.log("Soy un programa")  
}
```

← Declaración de la función

Qué podemos **llamar** siempre que queramos

```
saluda()  
saluda()  
  
// Hola  
// Soy un programa  
// Hola  
// Soy un programa
```

← Llamada a la función

← Llamada a la función

Funciones

Pueden recibir **parámetros**

```
function saluda(saludo) {  
  console.log(saludo)  
  console.log("Soy un programa")  
}
```

```
saluda("Buenas")
```

```
// Buenas
```

```
// Soy un programa
```

Funciones

Pueden recibir **varios parámetros** separados por comas

```
function saluda(saludo, presentacion){  
  console.log(saludo)  
  console.log(presentacion)  
}
```

```
saluda("Buenas", "Soy un algoritmo")
```

```
// Buenas
```

```
// Soy un algoritmo
```

Funciones

Las funciones **encapsulan** el código, lo que está dentro es inaccesible desde fuera

```
function saluda(saludo, presentacion){  
    let a = 1  
    console.log(saludo)  
    console.log(presentacion)  
}
```

← Solo tenemos acceso a los parámetros dentro del body
← Solo tenemos acceso a las variables dentro del body

```
saluda("Buenas", "Soy un algoritmo")  
console.log(saludo)  
console.log(presentacion)  
console.log(a)
```

```
// Buenas  
// Soy un algoritmo  
// error  
// error  
// error
```

Funciones

Pueden devolver un **valor** mediante un **return**

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

Ese valor lo podemos capturar con una variable

```
let resultado = suma(2, 3)  
console.log(resultado)
```

Funciones

Las **llamadas a funciones** son **expresiones**, por tanto podemos tratarlas como si fueran un valor:

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

```
console.log(suma(2, 3))  
// 5
```

Funciones

Las **llamadas a funciones** son **expresiones**, por tanto podemos tratarlas como si fueran un valor:

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

```
console.log(suma(2, 3))  
// 5
```

```
console.log(suma(suma(1, 1), suma(2, 2)))  
// 6
```

Ejercicio funciones

- Crea una función que reciba **tres** números por parámetro y devuelva la multiplicación de los tres números
- Llama a la función y comprueba que funciona imprimiendo en la consola

Ejercicio funciones II

- Crea una función que reciba un número por parámetro e imprima todos los números que hay entre ese número y cero de forma descendente.
- Llama a la función y comprueba que funciona

Ejercicio funciones III

- Crea una función que reciba dos números y devuelva el más grande

Estructuras de datos

Arrays

Arrays

Podemos crear cadenas de valores utilizando un **array** (símbolo [])

```
let cities = ["London", "Madrid", "Rome"]
```

Los arrays son valores de tipo **Object**

Arrays

Podemos declarar un **array vacío**

```
let cities = []
```

.push() añade valores a un array (de cualquier tipo)

```
cities.push(1)  
cities.push("Beta")  
cities.push(true)
```

```
console.log(cities)  
// [1, "Beta", true]
```

Arrays

Accedemos a una posición concreta del array especificando un **índice**

```
let cities = ["London", "Madrid", "Rome"]  
console.log(cities[0]) // London  
console.log(cities[1]) // Madrid
```

Arrays

Accedemos a una posición concreta del array especificando un **índice**

```
let cities = ["London", "Madrid", "Rome"]  
console.log(cities[0]) // London  
console.log(cities[1]) // Madrid
```

También podemos acceder a posiciones mediante **expresiones**

```
let i = 1  
console.log(cities[i]) // Madrid  
console.log(cities[1 + 1]) // Rome
```

Arrays

Modificar posiciones concretas de un array

```
let cities = ["London", "Madrid", "Rome"]  
cities[1] = "Berlin"  
console.log(cities) // ["London", "Berlin", "Rome"]
```


Arrays

.splice() permite eliminar segmentos del array

```
let cities = ["London", "Madrid", "Rome"]  
cities.splice(1,1)  
console.log(cities) // ["London", "Rome"]
```

splice recibe dos parámetros → splice(**índice**, **número de valores a cortar**)

Arrays

- splice recibe dos parámetros → splice(**índice**, **número de valores a cortar**)
- splice devuelve el fragmento recortado

```
let letters = ["a", "b", "c", "d"]
let subArray = letters.splice(1,2)
console.log(letters) // ?
console.log(subArray) // ?
```

Arrays

- splice recibe dos parámetros → splice(**índice**, **número de valores a cortar**)
- splice devuelve el fragmento recortado

```
let letters = ["a", "b", "c", "d"]
let subArray = letters.splice(1,2)
console.log(letters) // ["a", "d"]
console.log(subArray) // ["b", "c"]
```

Arrays

- splice recibe dos parámetros → splice(**índice**, **número de valores a cortar**)
- splice devuelve el fragmento recortado

```
let letters = ["a", "b", "c", "d"]  
let subArray = letters.splice(3,1)  
console.log(letters) // ?  
console.log(subArray) // ?
```

Arrays

- splice recibe dos parámetros → splice(**índice**, **número de valores a cortar**)
- splice devuelve el fragmento recortado

```
let letters = ["a", "b", "c", "d"]  
let subArray = letters.splice(3,1)  
console.log(letters) // ["a", "b", "c"]  
console.log(subArray) // ["d"]
```

Arrays

.length nos permite saber el número de valores en un array

```
let letters = ["a", "b", "c", "d"]  
console.log(letters.length) // 4
```

Ejercicio arrays

- Crea una función que reciba dos parámetros cualquiera y devuelva un array con esos dos parámetros dentro
- **Ejemplo:** parámetros: (10, 20) → devuelve un array: [10, 20]

Ejercicio arrays II

- Crea una función que reciba un número y devuelva una array con todos los números del 0 hasta ese número.
- Ejemplo: $4 \rightarrow [0, 1, 2, 3, 4]$

Ejercicio arrays III

- Crea una función que reciba un array y imprima uno a uno todos los valores de ese array

Ejercicio arrays IV

- Crea una función que reciba un array de números, la recorra e imprima la suma de todos los números

Ejercicio arrays V

- Crea una función que reciba un array de números y devuelva el más pequeño

Bucle for of

Bucle for of

Nos permiten recorrer todos los valores de un array de forma más sencilla

```
let letters = ["a", "b", "c", "d"]  
for(let letter of letters){  
    console.log(letter)  
}
```

```
// a  
// b  
// c  
// d
```

Ejercicio arrays VI - Bucle for of

- Crea una función que reciba un array de números y devuelva el más pequeño
- Utiliza un **bucle for**

Array nesting

Arrays

Un array puede contener **cualquier valor**, por tanto...

Un array puede contener **otro array**

```
[1, [2, 3], [4, 5, 6]]
```


Arrays

Esto nos permite, por ejemplo, representar filas y columnas

A1 A2

B1 B2

```
[["A1", "A2"], ["B1", "B2"]]
```

Arrays

```
[[2, 3, 8], [5, 6, 2]]
```

Cómo puedo imprimir uno a uno todos los números de esta **nested array**?

Demo

Arrays

```
[[2, 3, 8], [5, 6, 2]]
```

Cómo puedo imprimir uno a uno todos los números de esta **nested array**?

```
let array = [[2, 3, 8], [5, 6, 2]]
```

```
for(let i=0; i < array.length; i++){  
  let fila = array[i]  
  for(let i2=0; i2<fila.length; i++){  
    console.log(fila[i2])  
  }  
}
```

Ejercicio nested arrays

- Recorre y suma todos los números de este array utilizando **bucles for**:

- ```
let array = [[2, 2], [3, 4], [1, 1, 1]]
```

# Ejercicio nested arrays II

- Recorre y suma todos los números de este array utilizando **bucles for of**:

- ```
let array = [[2, 2], [3, 4], [1, 1, 1]]
```

Objetos

Objetos

Hemos visto que podemos usar **arrays** para **formar conjuntos** de datos

```
let letters = ["a", "b", "c", "d"]
```

Pero a veces no es la mejor opción:

```
let datosUsuario = ["javi88", "javier@gmail.com", "1234", 31, true, false]
```


Objetos

Los **objetos** `{}` son parecidos a las arrays, pero en vez de utilizar **índices** para guardar los valores utilizan **claves** (keys)

```
{clave: valor, clave: valor, ...}
```

Ejemplo

```
let datos = {"nombre": "Javier", "apellido": "Arrabal"}
```

Objetos

```
let datos = {"nombre": "Javier", "apellido": "Arrabal"}
```

Better **readability**: se suele escribir verticalmente

```
let  datos = {  
    "nombre": "Javier",  
    "apellido": "Arrabal"  
}
```

Objetos

Cómo accedemos a los valores de un objeto?

```
let datos = {  
  "nombre": "Javier",  
  "apellido": "Arrabal"  
}
```

Objetos

Accedemos a los valores mediante su **clave** de **dos formas distintas**

```
let datos = {  
  "nombre": "Javier",  
  "apellido": "Arrabal"  
}
```

```
console.log(datos["apellido"]) // Arrabal
```

```
console.log(datos.apellido) // Arrabal
```

Objetos

Añadimos valores mediante su **clave** de **dos formas distintas**

```
let datos = {  
  "nombre": "Javier",  
}
```

```
datos["apellido"] = "Arrabal"
```

```
datos.apellido = "Arrabal"
```

Objetos

Podemos añadir valores de **cualquier tipo** a un Objeto

```
let div = {  
  "colors": ["red", "blue"],  
  "dimensions": {"width": 400, "height": 300},  
  "remove": function() {  
    console.log("div has been removed")  
  }  
}
```

// ← Arrays
// ← Objetos
// ← Funciones

Objetos

```
let div = {  
  "colors": ["red", "blue"],  
  "dimensions": {"width": 400, "height": 300},  
  "remove": function() {  
    console.log("div has been removed")  
  }  
}
```

Cómo accedemos al color rojo?

Objetos

```
let div = {  
  "colors": ["red", "blue"],  
  "dimensions": {"width": 400, "height": 300},  
  "remove": function() {  
    console.log("div has been removed")  
  }  
}
```

Cómo accedemos al color rojo?

```
console.log(div.colores[0])
```


Objetos

```
let div = {  
  "colors": ["red", "blue"],  
  "dimensions": {"width": 400, "height": 300},  
  "remove": function() {  
    console.log("div has been removed")  
  }  
}
```

Cómo accedemos a la altura?

Objetos

```
let div = {  
  "colors": ["red", "blue"],  
  "dimensions": {"width": 400, "height": 300},  
  "remove": function() {  
    console.log("div has been removed")  
  }  
}
```

Cómo accedemos a la altura?

```
console.log(div.colores.height)
```

Objetos

```
let div = {  
  "colors": ["red", "blue"],  
  "dimensions": {"width": 400, "height": 300},  
  "remove": function() {  
    console.log("div has been removed")  
  }  
}
```

Cómo llamamos a la función?

Objetos

```
let div = {  
  "colors": ["red", "blue"],  
  "dimensions": {"width": 400, "height": 300},  
  "remove": function() {  
    console.log("div has been removed")  
  }  
}
```

Cómo llamamos a la función?

```
div.remove()
```

Ejercicio Objetos

- Declara un objeto con **tres conjuntos clave-valor** que guarde el nombre, la raza y el peso de un perro

Ejercicio Objetos II

- Modifica el objeto del ejercicio anterior para que el peso, en vez de ser un número, sea un objeto con **dos conjuntos clave valor**
 - Unidad (Ejemplo: kg, lbs, pounds, etc.)
 - Cantidad (Ejemplo: 7, 15, etc.)

Ejercicio Objetos III

- Vamos a crear una lista de la compra
- Declara una variable con un **array** que contenga **6 objetos**
- Cada objeto es un producto
- Cada objeto tiene **dos conjuntos clave-valor**:
 - Nombre
 - Precio

Ejercicio Objetos IV

- Recorre la lista de la compra del ejercicio anterior con un bucle e imprime los productos uno a uno

Ejercicio Objetos V

- Declara una **función** que reciba la lista de la compra del ejercicio anterior y la recorra con un bucle
- Calcula e imprime el **coste total** de todos los productos

Manipulando strings

Manipulando strings

Concatenar strings

```
let resultado = "Hello" + " " + "World"
```

```
console.log(resultado) // "Hello World"
```

Concatenar strings y números

```
let points = 7
```

```
console.log("Has conseguido: " + points + " puntos") // "Has conseguido: 7 puntos"
```

Manipulando strings

Longitud de una string

```
let string = "ABCDE"
```

```
console.log(string.length) //5
```

Manipulando strings

Encontrar el índice de una **substring** dentro de una string

```
let str = "Lorem ipsum"
```

```
str.indexOf("ipsum") // 6
```

Manipulando strings

Obtener una substring dado un índice

```
let str = "Lorem ipsum"
```

```
str.indexOf("ipsum") // 6
```

Devuelve -1 si no existe

```
let str = "Lorem lorem"
```

```
str.indexOf("ipsum") // -1
```

Manipulando strings

Podemos usarlo para detectar si una string contiene una substring

```
function contiene(str, substr){  
    let index = str.indexOf(substr)  
    return index >=0  
}
```

```
let texto = "lorem ipsum"  
console.log(contiene(texto, "ipsum")) // true  
console.log(contiene(texto, "amet"))  // false
```

Manipulando strings

Obtener una substring dado un índice

```
str.substring(startIndex, endIndex)
```

Ejemplos

```
let str = "Hello World"  
let result = str.substring(1, 6) // "ello W"
```

```
let str = "Hello World"  
let result = str.substring(6, str.length) // "World"
```


Manipulando strings

Separar una string mediante una substring intermedia

```
str.split(substring)
```

Ejemplos

```
let str = "lorem|ipsum|amet"  
str.split("|") // ["lorem", "ipsum", "amet"]
```

```
let str = "aa11bb111cc"  
str.split("111") // ["aa", "bb", "cc"]
```

Ejercicio string manipulation

- Crea una función que reciba una string y compruebe si está vacía

Ejercicio string manipulation II

- Crea una función que reciba una array con strings y devuelva la concatenación de todas las strings

Ejercicio string manipulation III

- Crea una función que reciba una string y compruebe si hay algún espacio

Ejercicio string manipulation IV

- Crea una función que reciba una frase y la separe en palabras (un array de palabras)

Ejercicio string manipulation V

- Crea una función que reciba una frase y cuente el número de espacios

Types

Hay diferentes tipos de variables (variable types)

- Numeros
- Strings
- Objetos
- Booleanos
- ...

Dynamic typing vs static typing

Dynamic typing

- Javascript es un lenguaje con **dynamic typing**
- No especificamos en ningún momento si una variable va a ser de un tipo o otro, Javascript lo deduce por nosotros

```
let x = 1  
x = 'abc'  
x = true  
x = {data: []}
```

Static typing

- Java, C++, Haskell son lenguajes con **static typing**
- Al crear una variable especifican su tipo y solo acepta valores de ese tipo

```
int x = 1;
```

```
String y = 'abc';
```

```
Boolean z = true;
```

Aunque no lo especifiquemos, las variables tienen tipo en Javascript

¿Por qué importa?

Static typing

```
let num1 = 20  
let num2 = "30"
```

```
console.log(num1 + num2)
```

- Cuál es el resultado?

Static typing

```
let num1 = 20
```

```
let num2 = "30"
```

```
console.log(num1 + num2)
```

- “2030”

Type coercion

Qué es type coercion?

- El proceso de convertir variables de un tipo a otro tipo

Coercion a números

- String -> Int

```
let x = parseInt('2019')
```

- String -> Float

```
let x = parseFloat('3.14')
```

Coercion a strings

- Numero -> String

```
let x = 2019.toString()
```

```
let x = (3.14).toString()
```

- Object -> String

```
let x = JSON.stringify( {status: "200"} )
```

- Array -> String

```
let x = JSON.stringify( [1, 2, 3] )
```

Ejercicio type coercion

- Crea una variable que contenga la string “3 20 6 1”
- Extrae los números de la string y conviértelos a tipo **number**
- Imprime el resultado de sumar todos los números

Intervalos

Intervalos

Javascript dispone de dos herramientas para gestionar intervalos de tiempo

- `setTimeout`
- `setInterval`

setTimeout

Ejecuta una función (llamada **callback function**) al cabo de **x** milisegundos

```
setTimeout(callback, x)
```

setTimeout

Ejecuta una función (llamada **callback function**) al cabo de **x** milisegundos

```
setTimeout(funcion, x)
```

Ejemplos

```
setTimeout(function() {  
    console.log("Ha pasado un segundo")  
}, 1000)
```

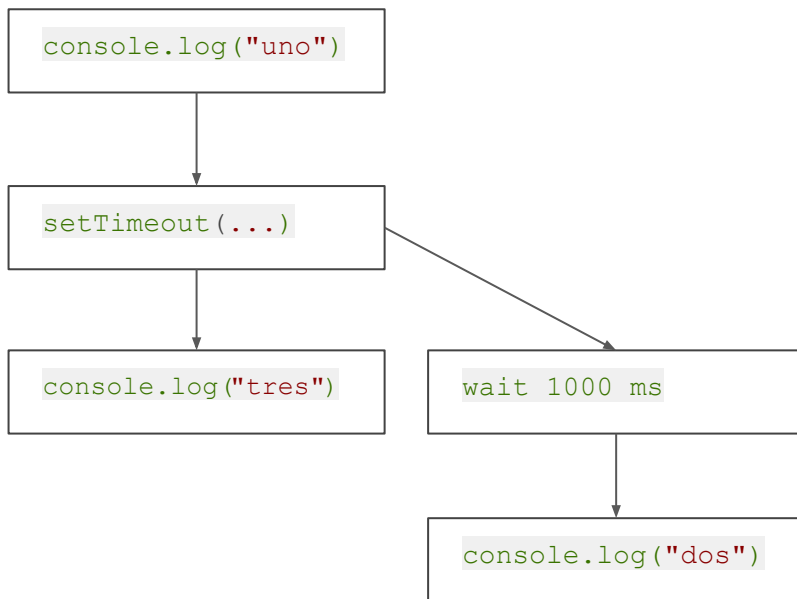
```
setTimeout(function() {  
    console.log("Ha pasado medio segundo")  
}, 500)
```


setTimeout

Genera **asincronía**: se ejecutan dos **hilos** de código por separado

```
console.log("uno")
setTimeout(function() {
  console.log("dos")
}, 1000)
console.log("tres")
```

```
// uno
// tres
// dos (al cabo de un segundo)
```



setInterval

Ejecuta una función **cada x** milisegundos

```
setInterval(funcion, x)
```

Ejemplos

```
setInterval(function() {  
    console.log("Me llamo cada vez que pasan dos segundos")  
}, 2000)
```

Ejercicio Intervalos

- Crea una variable contador que empiece valiendo 0
- Incrementa el contador cada segundo e imprímelo

Javascript y HTML

Document Object Model (DOM)

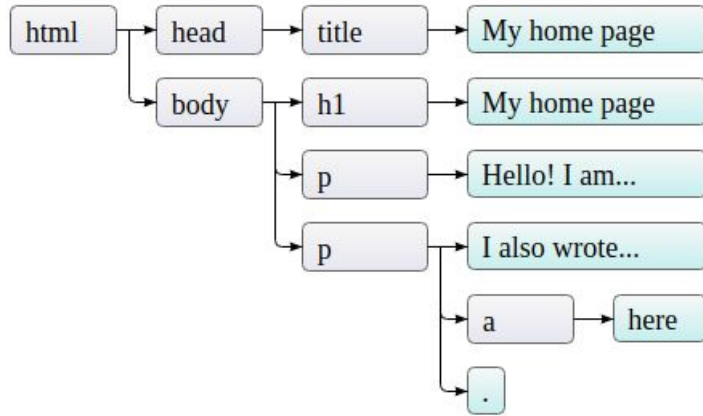
```
<!doctype html>
<html>
  <head>
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
    <p>Hello, I am Marijn and this is my home page.</p>
    <p>I also wrote a book! Read it.</p>
  </body>
</html>
```

Document Object Model (DOM)

```
<!doctype html>
<html>
  <head>
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
    <p>Hello, I am Marijn and this is my home page.</p>
    <p>I also wrote a book! Read it.</p>
  </body>
</html>
```

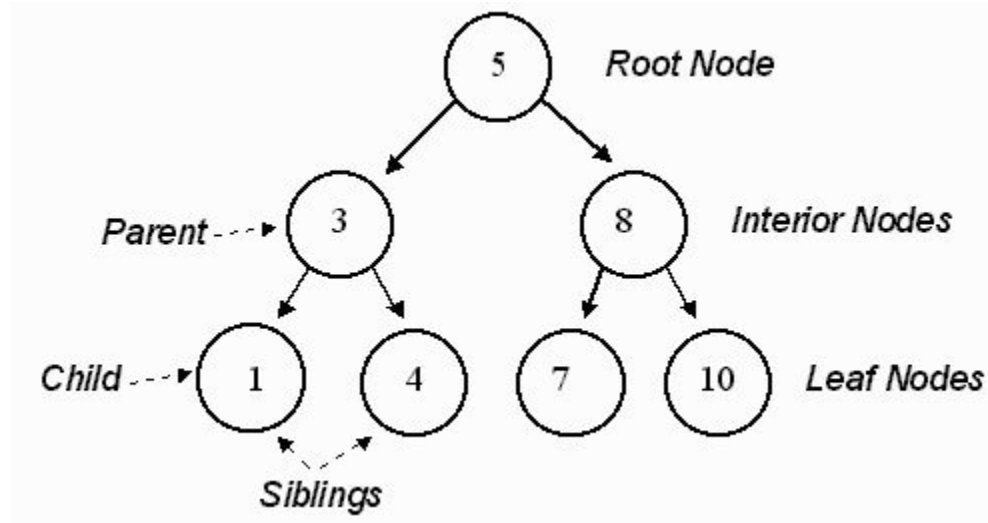
- En JS la variable **document** contiene el objeto que representa el DOM
- La variable document es accesible desde cualquier lado

Document Object Model (DOM)



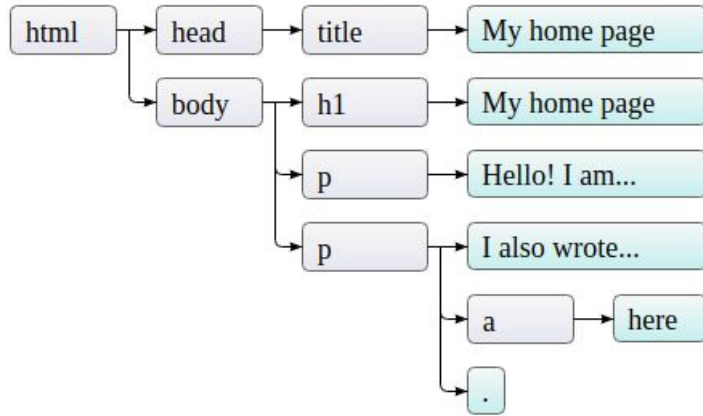
- El DOM tiene una estructura de **árbol**

Estructura de árbol



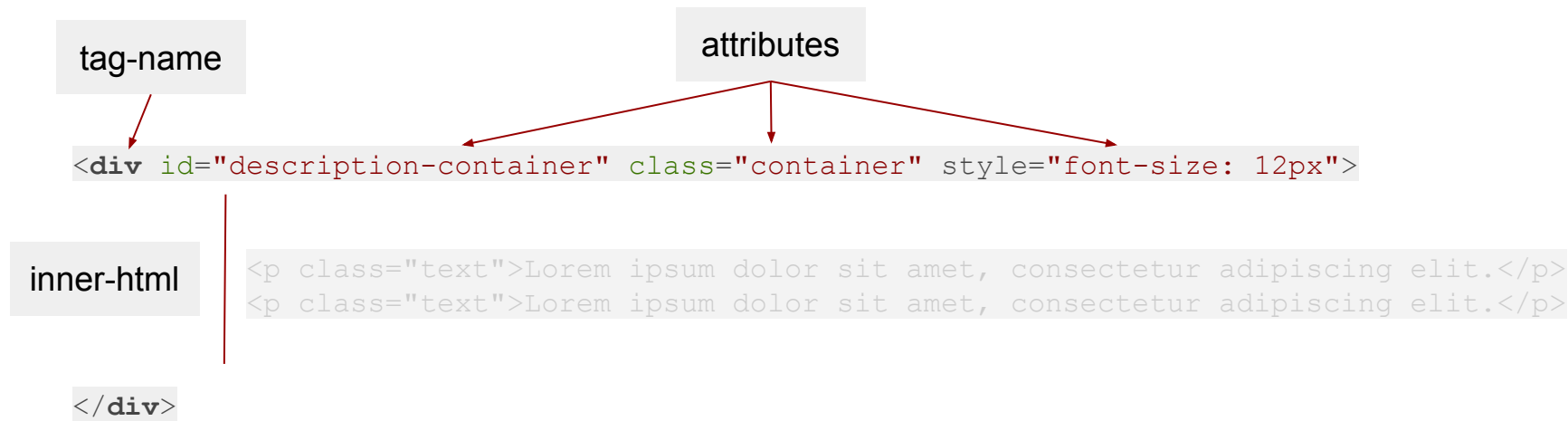
- Estructura jerárquica de padres e hijos

Document Object Model (DOM)



Anatomía de un nodo

Anatomía de un nodo



Selección de nodos

Selección un nodo

```
let node = document.querySelector('#node-id')
```

```
let node = document.querySelector('.node-class')
```

```
let node = document.querySelector('node-tag')
```

- Nos guarda el objeto del nodo en una variable
- Los mismos selectores (id, class, tag) que **CSS**

Selección de múltiples nodos

```
let node = document.querySelector('#node-id')
```

```
let nodes = document.querySelectorAll('.node-class')
```

```
let nodes = document.querySelectorAll('node-tag')
```

- **querySelectorAll** devuelve un grupo de nodos
- Seleccionar múltiples nodos a partir de un id no tiene sentido

Selección de múltiples nodos

```
let nodes = document.querySelectorAll('node-tag')
```

- nodes será una variable de tipo... ?

Selección de múltiples nodos

```
let nodes = document.querySelectorAll('node-tag')
```

- Array

Selección de múltiples nodos

```
let nodes = document.querySelectorAll('node-tag')
```

- ~~Array~~ \longrightarrow **NodeList**

Selección de múltiples nodos

```
let nodes = document.querySelectorAll('node-tag')
```

- Array \longrightarrow NodeList

foreach, map, filter, reduce, slice, splice...

Solución: type coercion

nodeList -> Array

Selección de múltiples nodos

```
let nodes = document.querySelectorAll('node-tag')  
nodes = [...nodes]
```

Alternativamente:

```
let nodes = [...document.querySelectorAll('node-tag')]
```

- El spread operator se encarga del type coercion
- Ahora ya podemos llamar a los métodos de array en nodes

Otros métodos de selección

```
let node = document.getElementById("node-id")  
let node = document.getElementsByClassName("node-class")  
let node = document.getElementsByTagName("node-tag")
```

```
let nodes = document.getElementsByClassName("node-class")  
let nodes = document.getElementsByTagName("node-tag")
```

Ejercicio selección

- Ruta ejercicio: (pendiente)
- Selecciona los elementos **rojos** utilizando **querySelector**
- Elimínalos utilizando el método **remove()** en los nodos

Ejercicio selección II

- Ruta ejercicio: (pendiente)
-
- Selecciona los elementos **verdes** utilizando los métodos **getElementBy**
- Elimínalos utilizando el método **remove()** en los nodos

Hemos seleccionado nodos, qué
podemos hacer con ellos?

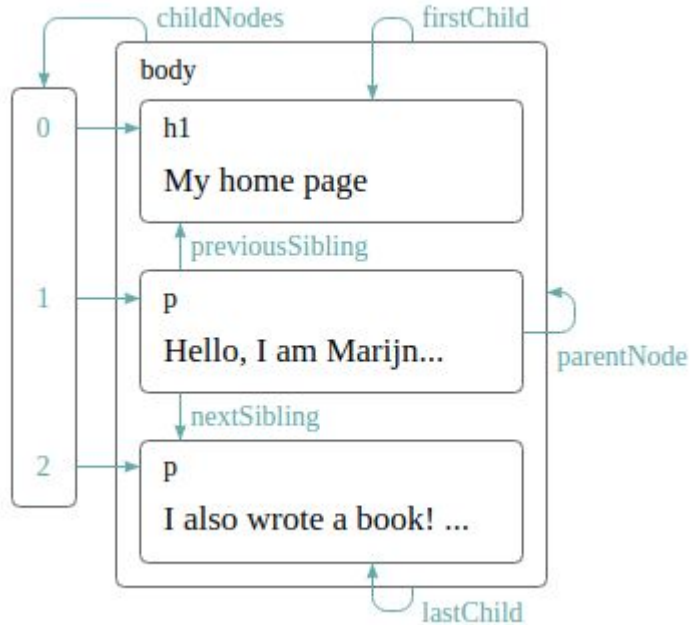
Nodos

```
let node = document.querySelector('#node-id')
```

- Los nodos tienen una serie de métodos que podemos usar para:
 - Recorrer el árbol
 - Modificar el árbol
 - Modificar los atributos y estilos (CSS) del nodo
 - Capturar las acciones del usuario sobre los nodos

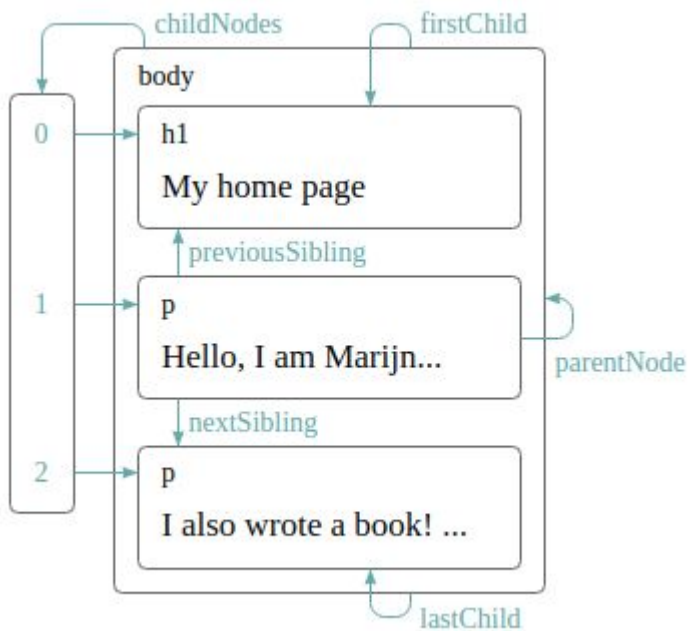
Movimiento a través del árbol

Propiedades de movimiento



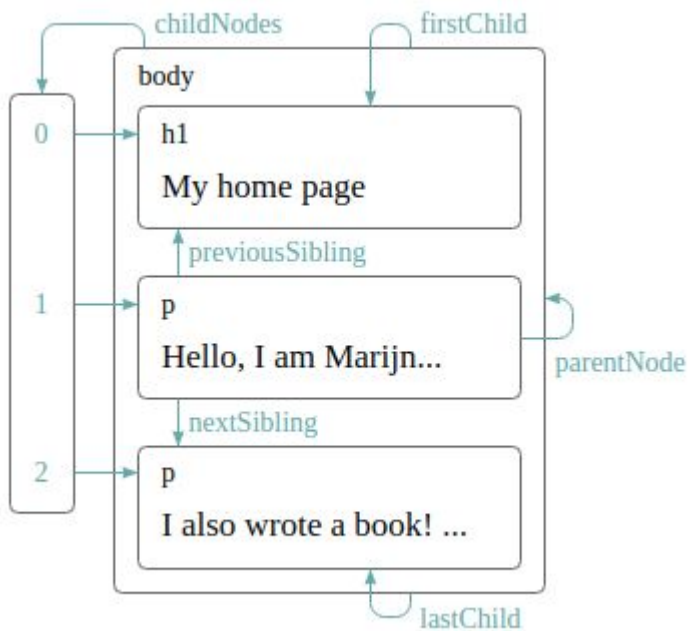
- No son métodos, son propiedades

Propiedades de movimiento



```
let parent = node.parentNode
let child = node.firstChild
let child = node.lastChild
let children = node.childNodes
let sibling = node.previousSibling
let sibling = node.nextSibling
```

Propiedades de movimiento



```
let parent = node.parentNode
```

```
let child = node.firstChild
```

```
let child = node.lastChild
```

```
let children = node.childNodes
```

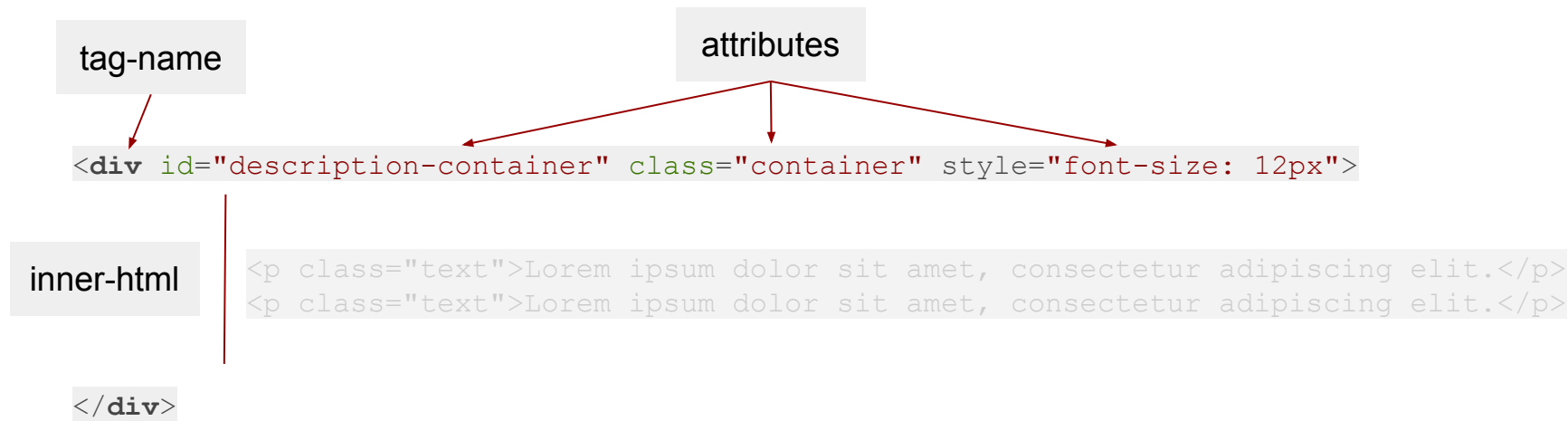
```
let sibling = node.previousSibling
```

```
let sibling = node.nextSibling
```

nodeList!

Creación de un nodo

Anatomía de un nodo



Creación de un nodo

```
let node = document.createElement(tagName)
```

- **document.createElement** nos permite crear un nodo
- **tagName** determina el tipo de nodo
- El nodo está guardado en la variable, pero **todavía no está en el DOM**

Creación de un nodo

```
let div = document.createElement("div")
let span = document.createElement("span")
let a = document.createElement("a")
let p = document.createElement("p")
let li = document.createElement("li")
let ul = document.createElement("ul")
let img = document.createElement("img")
let select = document.createElement("select")
```

Asignación de atributos

```
node.setAttribute(key, value)
```

```
// ejemplo:
```

```
node.setAttribute("href", "https://www.google.com");
```

- **setAttribute** asigna el valor de un atributo
- La asignación se realiza mediante clave/valor

Asignación de atributos

```
node.setAttribute(key, value)
```

```
// ejemplos:
```

```
node.setAttribute("href", "https://www.google.com");  
node.setAttribute("type", "button");  
node.setAttribute("id", "container");  
node.setAttribute("src", "/js/main.js");  
node.setAttribute("style", "width: 400px;");  
node.setAttribute("class", "text-container big-container");
```

Asignación de atributos

```
node.setAttribute(key, value)
```

```
// ejemplos:
```

```
node.setAttribute("href", "https://www.google.com");
```

```
node.setAttribute("type", "button");
```

```
node.setAttribute("id", "container");
```

```
node.setAttribute("src", "/js/main.js");
```

```
node.setAttribute("style", "width: 400px;");
```

```
node.setAttribute("class", "text-container big-container");
```

```
// better way
```

```
// better way
```

Lectura de atributos

```
let value = node.getAttribute(key)
```

```
// ejemplos:
```

```
let href = node.getAttribute("href")
```

```
let id = node.getAttribute("id")
```

```
// https://www.google.com
```

```
// container
```

Lectura de atributos

En el caso de los elementos **input** utilizamos la propiedad **value**

```
<input type="text" id="usuario" name="usuario">
```

```
let node = document.querySelector("usuario")  
console.log(node.value)
```

Hemos creado nodos, ¿como los
hacemos visibles?

Modificación del árbol

Inyección de nodos

```
let node = document.createElement(tagName)  
parent.appendChild(node)
```

- Los nodos se adjuntan siempre a un padre
- Ese nodo pasará a ser hijo del padre y estar contenido dentro

Inyección de nodos

```
// ejemplo 1
```

```
let div = document.createElement("div")  
let parent = document.querySelector("#container")  
parent.appendChild(node)
```

- Seleccionamos el padre y le adjuntamos un hijo
- Qué pasa si no hay ningún nodo en el body?

Inyección de nodos

```
// ejemplo 2
```

```
let div = document.createElement("div")  
document.body.appendChild(node)
```

- Podemos adjuntar nodos directamente al body

Movimiento de nodos

- [appendChild](#)
- [insertBefore](#)
- [insertAfter](#)

Eliminación de un nodo

```
let node = document.createElement(tagName)  
node.remove()
```

Ejercicio DOM

- Crea un nodo de tipo `a` usando javascript
- Añadele un href a una página web
- Insertalo en el DOM
- Comprueba que el enlace funciona

Ejercicio DOM II

- Crea un div que contenga el texto “color:” usando javascript
- Debajo: crea un nodo de tipo select con cinco nodos option (cinco colores) usando javascript

User input

User input

- Recibimos las acciones del usuario en forma de eventos
- Algunos ejemplos:
 - Click
 - Scroll
 - Pasar el ratón por encima
 - Pulsar una tecla
 - Introducir texto en un campo
 - Seleccionar una opción en un select

Event Listeners

- Para gestionar que sucede cuando ocurren estos eventos usamos **Event Listeners**
- Algunos ejemplos:
 - onclick
 - onmouseover
 - onmouseout
 - onchange (input)
 - onkeypress
 - onkeyup
 - onkeydown

Event Listeners

- El event listener recibe un callback
- El event listener queda a la espera de que suceda el evento
- Cuando sucede, llama al callback pasándole un objeto **event**

```
let node = document.querySelector( '#button' )
```

```
node.onclick = clickCallback
```

```
function clickCallback(event) {  
    console.log('has pulsado el botón' )  
}
```

Event Listeners

- El event listener recibe un callback
- El event listener queda a la espera de que suceda el evento
- Cuando sucede, llama al callback pasándole un objeto **event**

```
let node = document.querySelector( '#button' )
```

```
node.addEventListener( 'click', clickCallback )
```

```
function clickCallback( event ) {  
    console.log( 'has pulsado el botón' )  
}
```

Event Listeners

- El event listener recibe un callback
- El event listener queda a la espera de que suceda el evento
- Cuando sucede, llama al callback pasándole un objeto **event**

```
<div onclick="clickCallback()"></div>
```

```
function clickCallback() {  
    console.log('has pulsado el div')  
}
```

Event Listeners

- Los event listeners se pueden borrar

```
node.addEventListener('click', clickCallback)
```

```
function clickCallback(event) {  
  console.log('has pulsado el div')  
}
```

Event Listeners

- Los event listeners se pueden borrar

```
node.removeEventListener('click', clickCallback)
```

```
function clickCallback(event) {  
    console.log('has pulsado el div')  
}
```

Event Listeners

- Los event listeners se pueden borrar

```
node.onclick = clickCallback
```

```
function clickCallback(event) {  
    console.log('has pulsado el div')  
}
```


Event Listeners

- Los event listeners se pueden borrar

```
node.onclick = undefined
```

Ejercicio Event listeners

- Crea un nodo **div** usando javascript
- Añadele un texto
- Añádele un event listener para que imprima “Pressed” al ser pulsado

Ejercicio Event listeners II

- Crea un nodo **div** usando javascript
- Añadele un texto
- Añádele un event listener para que cambie el color del texto al pasar el ratón por encima y vuelva a la normalidad al apartar el ratón

Objeto event

El objeto event

- El objeto **event** que reciben los callbacks tiene información sobre el evento
- Algunos ejemplos
 - Qué boton del ratón se ha pulsado
 - Qué nodo ha lanzado el evento (**target**)
 - Qué tipo de evento es (click, mousedown, keypress, etc.)
 - Qué tecla se ha pulsado
 - Coordenadas del ratón
 - Información sobre scroll del ratón

Ejemplo click

```
// event.button
```

```
let div = document.createElement("div")  
let parent = document.querySelector("#container")  
parent.appendChild(div)
```

```
div.onclick = clickHandler
```

```
function clickHandler(event) {  
    if(event.button === 0) { console.log("click izquierdo") }  
    if(event.button === 1) { console.log("click derecho") }  
}
```

Ejemplo target

```
// event.target
```

```
let div = document.createElement("div")
let p = document.createElement("p")
let parent = document.querySelector("#container")
parent.appendChild(div)
parent.appendChild(p)
```

```
div.onclick = clickHandler
p.onclick = clickHandler
```

```
function clickHandler(event) {
  console.log("Hover en " + event.target.tagName)
}
```

Ejemplo coordenadas

```
// event.pageX event.pageY
```

```
let div = document.createElement("div")  
let parent = document.querySelector("#container")  
parent.appendChild(div)
```

```
div.onmouseover = mouseOverHandler
```

```
function mouseOverHandler(event) {  
    console.log("Hover en " + event.target.tagName)  
    console.log("Coordenadas: ", event.pageX, event.pageY)  
}
```


Ejercicio event object

- Crea 3 divs con diferente id
- Cada div debe imprimir su id al ser pulsado
- Utiliza solo **1** clickHandler

Ejercicio event object

- Recupera el código del ejercicio DOM II
- Cambia el color del texto “color:” en función del color que selecciones en el select

Ejercicio event object II

- Crea el código necesario para que al pulsar la tecla enter se imprima “Enter pressed”
- Utiliza [StackOverflow](#) o lee [documentación](#) para descubrir cómo hacerlo

Scroll

Scroll

El event listener **onscroll** captura el scroll del usuario

```
document.onscroll = function() {  
    console.log('se ha hecho scroll')  
}
```

Scroll

Podemos saber el **número de píxeles** avanzados actualmente en la página mediante la propiedad **scrollTop**

```
let ammount = document.documentElement.scrollTop  
let ammount = document.body.scrollTop
```

```
//350px
```

```
//350px (navegadores antiguos)
```

Y forzar el nivel de scroll

```
document.documentElement.scrollTop = 1000
```

Scroll

scrollTop se puede utilizar con cualquier elemento contenedor

```
let ammount = node.scrollTop  
node.scrollTop = 1000
```

Scroll

Es fácil convertir píxeles de scroll a porcentaje

```
let ammount = document.documentElement.scrollTop //350px  
let perc = 100 * ammount / document.body.scrollHeight //24.681
```


Ejercicio scroll

- Crea un div con posición fixed
- El div debe cambiar de color una vez se supera el 50% de scroll
- El div debe volver al color inicial si el scroll disminuye por debajo del 50%

Regular expressions

Regex

Las **regex** son herramientas que nos permiten manipular strings de una forma muy **versátil** y **eficiente**

Casi todos los lenguajes de programación implementan **expresiones regulares**

Regex

Formulando una expresión regular podemos

- Buscar texto en una string
- Reemplazar substrings
- Extraer información de una string

Pero lo más importante es que esas búsquedas pueden ser muy específicas

Declaración

Es posible **declarar** expresiones regulares de dos formas

```
const re = new RegExp('abc')
```

```
const re = /abc/           // regex literal
```

Test

Podemos *testear* una regex contra una string

```
re.test(string)
```

Ejemplos

```
const re = /abc/  
re.test('111abc111') //true
```

```
/abc/.test('abracadabra') //true  
/abc/.test('cba') //false
```

Anchoring

Comprobar que empieza por una string (^)

```
/^abc/.test('abc 111')  
/^abc/.test('111 abc')
```



Comprobar que termina en una string (\$)

```
/abc$/.test('abc 111')  
/abc$/.test('111 abc')
```



Anchoring

Comprobar que la Regex coinciden completamente

```
// ?
```


Anchoring

Comprobar que la Regex coinciden completamente

```
/^abc$/.test('abc')  
/^abc$/.test('abc 111')  
/^abc$/.test('111 abc')
```

```
// ✓  
// ✗  
// ✗
```

Match item in range

Comprobar que en una string aparece algún carácter de un rango

```
/[a-z]/ //a, b, c, ... , x, y, z  
/[A-Z]/ //A, B, C, ... , X, Y, Z  
/[a-c]/ //a, b, c  
/[0-9]/ //0, 1, 2, 3, ... , 8, 9
```

```
/[a-z]/.test('a') //✓  
/[a-z]/.test('1') //✗  
/[a-z]/.test('A') //✗
```

```
/[a-c]/.test('d') //✗  
/[a-c]/.test('dc') //✓
```

Match item in range





Podemos combinar rangos

```
/[A-Za-z0-9]/
```

```
/[A-Za-z0-9]/.test('a') //✓  
/[A-Za-z0-9]/.test('1') //✓  
/[A-Za-z0-9]/.test('A') //✓
```

Match negate item in range

Cuando está dentro de un rango, (^) niega

```
/[^A-Za-z0-9]/.test('a') //   
/[^A-Za-z0-9]/.test('1') //   
/[^A-Za-z0-9]/.test('A') //   
/[^A-Za-z0-9]/.test('@') // 
```

OR

Operación lógica OR (|)

```
/lorem|ipsum/.test('lorem') //✓  
/lorem|ipsum/.test('ipsum') //✓
```

Metacharacters

Las regex acepta metacharacters. Caracteres con diferentes significados

- [Referencia](#)

Metacharacters

Imaginemos que queremos determinar si una string esta formada por un solo dígito

Podemos usar el metacaracter `\d`

```
/^ \d$/.test('1')    //✓  
/^ \d$/.test('11')   //✗
```

Si queremos comprobar que haya más de un dígito podemos utilizar **quantifiers**

Quantifiers

Una o más veces (+)

```
/^\d+$/
```

```
 /^\d+$/ .test('12')    // ✓  
 /^\d+$/ .test('14')    // ✓  
 /^\d+$/ .test('144343') // ✓  
 /^\d+$/ .test('')      // ✗  
 /^\d+$/ .test('1a')    // ✗
```


Quantifiers

Cero o más veces (*)

```
/^\d*$/
```

```
 /^\d*$/ .test('12')    // ✓  
 /^\d*$/ .test('14')    // ✓  
 /^\d*$/ .test('144343') // ✓  
 /^\d*$/ .test('')      // ✓  
 /^\d*$/ .test('1a')    // ✗
```

Quantifiers

Entre **n** y **m** veces (**{n, m}**)

```
/^\d{3,5}$/
```

```
/^\d{3,5}$/.test('123')    //✓  
/^\d{3,5}$/.test('1234')   //✓  
/^\d{3,5}$/.test('12345')  //✓  
/^\d{3,5}$/.test('123456') //✗
```

Quantifiers

Entre **n** o más veces (**{n,}**)

```
/^\d{3,}$/
```

```
//^\d{3,}$/ .test('12') //✗  
//^\d{3,}$/ .test('123') //✓  
//^\d{3,}$/ .test('12345') //✓  
//^\d{3,}$/ .test('123456789') //✓
```

Quantifiers

Entre **n** o más veces (**{n,}**)

```
/^\d{3,}$/
```

```
//^\d{3,}$/ .test('12') //✗  
//^\d{3,}$/ .test('123') //✓  
//^\d{3,}$/ .test('12345') //✓  
//^\d{3,}$/ .test('123456789') //✓
```

Quantifiers

Condiciones opcionales con ?

```
/^\d{3}\w?$/
```

```
/^\d{3}\w?$/ .test('123')    // ✓  
/^\d{3}\w?$/ .test('123a')   // ✓  
/^\d{3}\w?$/ .test('123ab')  // ?
```

Quantifiers

Condiciones opcionales con ?

```
/^\d{3}\w?$/
```

```
/^\d{3}\w?$/ .test('123') //✓
```

```
/^\d{3}\w?$/ .test('123a') //✓
```

```
/^\d{3}\w?$/ .test('123ab') //✗ 3 o más caracteres, seguidos de un caracter opcional
```

```
???.test('123ab') //✓
```

Quantifiers

Condiciones opcionales con ?

```
/^\d{3}\w?$/
```

```
/^\d{3}\w?$/ .test('123') //✓
```

```
/^\d{3}\w?$/ .test('123a') //✓
```

```
/^\d{3}\w?$/ .test('123ab') //✗ 3 o más caracteres, seguidos de un caracter opcional
```

```
/^\d{3}\w*$/ .test('123ab') //✓
```

Grupos

Podemos crear **grupos** de caracteres usando paréntesis




```
/^(\d{3})(\w+)$/
```

```
/^(\d{3})(\w+)$/.test('123')           // ❌  
/^(\d{3})(\w+)$/.test('123s')          // ✔️  
/^(\d{3})(\w+)$/.test('123something')  // ✔️  
/^(\d{3})(\w+)$/.test('1234')          // ✔️
```


Grupos

Los **quantifiers** detrás de un grupo afectan al conjunto del grupo

```
/^(\d{2})+$/
```

```
/^(\d{2})+$/ .test('12')    //   
/^(\d{2})+$/ .test('123')   //   
/^(\d{2})+$/ .test('1234')  // 
```

Capturando grupos

Podemos comprobar qué substring ha coincidido con cada grupo utilizando

```
String.match(RegExp)  
RegExp.exec(String)
```

Ambos métodos devuelven un array:

```
[coincidencia global, coincidencia grupo1, coincidencia grupo2, ...]
```

```
'--123s--'.match(/^(\d{3})(\w+)$/) // [ "123s", "123", "s" ]
```

```
/^(\d{3})(\w+)$/.exec('--123s--') // [ "123s", "123", "s" ]
```

Capturando grupos

De la misma forma, podemos saber **qué forma tiene** la substring que ha coincidido con una regex

```
/^\d{3}\w+$/ .exec('--123s--')[0]
```

```
// "123s"
```

Flags

Podemos configurar las regex con opciones que llamamos **flags** o **modifiers**

Case insensitive (**i**)

```
/hey/i.test('HEy') //✓
```

Global (**g**): busca múltiples instancias de la substring

```
/hey/ig.test('HEy') //✓
```

Replace

Podemos utilizar las regex para **substituir** substrings

```
string.replace(regex, substitución)
```

Ejemplos:

```
"Hello world!".replace(/world/, 'dog')
```

```
//Hello dog!
```

```
"My dog is a good dog!".replace(/dog/, 'cat')
```

```
//My cat is a good dog!
```

```
"My dog is a good dog!".replace(/dog/g, 'cat')
```

```
//My cat is a good cat!
```

Ejercicio Regex

- Crea una regex que extraiga un número de un array cualquiera
 - Ejemplo `'Test 123123329'`
- La string puede tener cualquier otra forma

Ejercicio Regex II

- Crea una regex que compruebe que un correo electrónico es valido
 - Ejemplo `'ejemplo@gmail.com'`
- Comprueba que se cumple esta secuencia
 - Texto
 - @
 - Texto
 - .
 - Texto

Ejercicio Regex III

- Crea una regex que extraiga texto entrecomillado con comillas dobles
- Ejemplo: `'Texto: "captura esto"'`
- Solo si el texto objetivo no tiene ningún carácter (‘)