

# LIND-2C Programación Web

## Javascript

Mg. Pablo Ezequiel Inchausti  
Ing. David Túa

UNIVERSIDAD DEL CEMA  
**UCEMA**

---

Jueves 31 de Agosto, 09:00 hs

## Tercer paso... ¡la vida!

---

Ya sabemos desarrollar una página estática: tiene forma, estilo y contenido, pero este último no puede ser modificado. Además, la posibilidad de interactuar con la aplicación es mínima. Para lograr la funcionalidad de una aplicación completa aún nos resta aprender Javascript, el llamado “lenguaje de internet”. Si bien no es un lenguaje exclusivo para el frontend, su vasta popularidad es debida principalmente a que es el lenguaje interpretado por los navegadores. Comenzaremos entonces conociendo sus fundamentos.



# ¿Qué es Javascript? Conceptos generales



- ☐ Javascript, a diferencia de HTML y CSS, es un lenguaje de programación
- ☐ Es un dialecto del estándar ECMAScript
- ☐ Es un lenguaje de programación interpretado, imperativo, débilmente tipado y dinámico
- ☐ Originalmente estuvo basado en prototipos, pero con el avance de las sucesivas versiones fue acercándose a un lenguaje de programación orientado a objetos
- ☐ Está en constante evolución vía nuevas versiones ECMAScript (ESx)
- ☐ Todos los navegadores saben interpretar Javascript y de allí su vasta popularidad
- ☐ Los archivos tienen usualmente extensión js
- ☐ En el archivo html, se especificará la ubicación de los distintos programas Javascript que podremos ejecutar
- ☐ Se pueden escribir con cualquier programa editor de textos. Nosotros utilizaremos un IDE

# Dinámica y débilmente tipado

Javascript es un lenguaje tanto dinámicamente tipado como débilmente tipado. Aprendamos qué significa cada una de dichas catalogaciones.



- Dinámicamente tipado: en Javascript no es necesario definir el tipo de dato que recibirá una variable, ya que el mismo se definirá en tiempo de ejecución cuando se asigna un valor por primera vez. A su vez, durante la ejecución del programa, podrá cambiar de tipo de dato simplemente asignando un valor de otro tipo de dato.

```
1 a=5
2 console.log(typeof a)
3 a="5"
4 console.log(typeof a)
```

number  
string

- Débilmente tipado: a diferencia de otros lenguajes de programación, Javascript realiza conversiones automáticas del tipo de dato cuando debe realizar operaciones con dos tipos de datos incompatibles. Por ejemplo, si debe sumar un dato numérico y uno alfanumérico, simplemente realiza una conversión automática del dato numérico y los suma como si ambos fuesen alfanuméricos.

```
1 a=1
2 b="2"
3 console.log(a+b)
```

12

# Tipos de datos

Entonces, ¿qué **tipos de datos** podemos utilizar en Javascript? Comencemos con los tipos de datos primitivos:

Tipo de Dato	¿Qué contiene?	Ejemplos
<b>Number</b>	Un valor numérico (con o sin decimales)	10 / 3.14159
<b>String</b>	Una cadena de caracteres	Este es un ejemplo de tipo string
<b>Boolean</b>	Un valor booleano: verdadero o falso	true / false
<b>Bigint</b>	Un valor numérico entero muy grande	9007199254740992
<b>Undefined</b>	Una variable a la que aún no se le ha asignado valor	Undefined
<b>Symbol</b>	Un valor único e inmutable	Se utiliza generalmente como clave de una propiedad de un objeto

A su vez, existen otros tipos de datos no primitivos: **Null**, **Object** y **Function**, pero aprenderemos su utilidad más adelante.

# Asignación de valores a variables

---

Al igual que en Python, la asignación de valores la realizaremos mediante el operador “=”.

```
1  cantidad = 28
2
```

En dicha inicialización, podremos utilizar en forma optativa la palabra “**var**” para decirle a Javascript que estamos definiendo una nueva variable.

```
1  var cantidad = 28
2
```

Como vemos, no debemos definir qué tipo de dato contendrá la variable “cantidad”, porque como ya aprendimos, Javascript es un programa dinámicamente tipado y define el tipo de dato durante la ejecución en la primer asignación.

# Impresión por consola y comentarios

Si quisiéramos imprimir por consola algún mensaje o contenido de variables, deberemos utilizar `console.log`.

```
1 console.log("Mostrando por consola...")
2 var nombre = "Sergio"
3 var apellido = "Sirotinsky"
4 console.log('Soy', nombre + " " + apellido)
5
```

```
Mostrando por consola...
Soy Sergio Sirotinsky
```

Además, si quisiéramos incorporar comentarios en el código, podremos hacerlo de la siguientes formas:

```
1 // A continuación vamos a mostrar varias cosas por consola
2 console.log("Mostrando por consola...")
3 var nombre = "Sergio"
4 var apellido = "Sirotinsky"
5 /*
6 Finalmente, mostramos un ejemplo donde
7 no sólo mostramos un literal sino también
8 el contenido de varias variables concatenadas
9 */
10 console.log('Soy', nombre + " " + apellido)
11
```

De esta forma insertamos un comentario de línea única

De esta forma insertamos un comentario de varias líneas

# Punto y coma final

---

El estándar de Javascript nos pide incorporar un **punto y coma “;”** luego de cada instrucción. Sin embargo, si nos olvidáramos de ponerlo, el interprete de Javascript realizará esta tarea por nosotros en el momento de la ejecución del programa.

Cabe destacar que no todas las instrucciones deberán finalizar con punto y coma, sino sólo aquellas que sean terminales. Por ejemplo, si estamos comenzando un ciclo repetitivo o una bifurcación (mediante “if” y “for”, instrucciones que veremos más adelante), no deberemos utilizar punto y coma.

Anteriormente estaba considerada una mala práctica el no uso de punto y coma, ya que se delegaba esta tarea en el interprete y por tanto su rendimiento decaía. Hoy en día se sabe que esta tarea es menor y no implica un esfuerzo adicional por parte del interprete, y por lo tanto se ven muchos programas sin el uso de punto y coma.

En términos generales, por lo tanto, el uso de punto y coma resulta indistinto. Sin embargo, y es importante saberlo, existen ciertos casos de borde donde el interprete pondrá punto y coma en lugares donde nosotros no lo hubiésemos puesto y, de esta manera, el programa tendrá un funcionamiento distinto al que nosotros hubiésemos querido.



# Template strings

Una mejora de la versión ES6 de Javascript permitió incorporar los **template strings**. Esto implica el formateo de una cadena de caracteres utilizando, por ejemplo, valores de variables. Conozcamos su funcionamiento.

```
1 var nombre = "Sergio"
2 var apellido = "Sirotinsky"
3 console.log(`Hola! Me llamo ${nombre} ${apellido}`)
4
```

Hola! Me llamo Sergio Sirotinsky

En el ejemplo se visualiza como se reemplaza el contenido de las variables en el texto descripto.

```
1 var radio = 10
2 var pi = 3.14159
3 console.log(`Longitud circunferencia: ${pi * radio * 2}`)
4
```

Longitud circunferencia: 62.8318

En este caso, vemos que no sólo es posible reemplazar lo que está dentro de las llaves por el contenido de una variable, sino también por cualquier código de ejecución válido para JavaScript, por ejemplo una cuenta.

En Javascript las bifurcaciones se realizan mediante la instrucción “if” y “else”.

```
1 var edad = 25;
2 if (edad >= 18) {
3     console.log('Es mayor de edad')
4 } else {
5     console.log('Es menor de edad')
6 }
7
```

Es mayor de edad

Veamos algunos detalles sobre el uso del “if”:

- La **condición** debe ir **entre paréntesis**, en este caso la edad debe ser mayor o igual a 18
- Las **instrucciones** que se ejecutan cuando la condición es verdadera o falsa, deberán estar **enmarcadas entre las llaves “{” y “}”**. En el caso del ejemplo hay una única instrucción, pero es posible incorporar más de una.
- Debido al uso de las llaves, la **tabulación** pasa a ser **optativa** en Javascript, a diferencia de Python donde era obligatoria.

# Bifurcaciones

Si quisieramos incorporar bifurcaciones encadenadas, deberemos utilizar **“else if”**, análogamente al **“elif”** de Python.

```
1  var nota = 3
2  v if (nota == 10) {
3      console.log('Sobresaliente')
4  v } else if (nota >= 8) {
5      console.log('Superado')
6  v } else if (nota >= 6) {
7      console.log('Aprobado')
8  v } else if (nota >= 4) {
9      console.log('Recupera')
10 v } else {
11     console.log('Reprobado')
12 }
13
```

Reprobado

# Condiciones

Hemos visto que, al igual que en Python, deberemos utilizar una condición para determinar la bifurcación. En Javascript dichas condiciones se escriben en forma casi idéntica a Python pero con una leve diferencia: en el caso en que debamos incorporar un operador lógico para conectar dos expresiones condicionales, deberemos utilizar los siguientes:

- && en lugar del “and”, cuando queremos ejecutar sólo cuando ambas condiciones sean verdaderas
- || en lugar del “or”, cuando queremos ejecutar cuando cualquiera de las condiciones son verdaderas

```
1  a=5
2  b=3
3  v if (a==5 && b==3){
4      console.log('se imprime AND!')
5  }
6  v if (a==4 && b==3){
7      console.log('no se imprime AND!')
8  }
9  v if (a==4 || b==3){
10     console.log('se imprime OR!')
11 }
12
```

```
se imprime AND!
se imprime OR!
```

# Ciclos repetitivos. While

Al igual que en Python, existen dos tipos de **ciclos repetitivos**. Aquellos donde conocemos de antemano la cantidad de veces que queremos iterar y aquellos en donde no se conoce pero depende del cumplimiento de una determinada condición. Veamos entonces ejemplos para ambos casos.

Si quisiéramos repetir una serie de instrucciones mientras una condición es verdadera, deberemos utilizar el **“while”**.

```
1  monto = 50;  
2  while (monto < 1000) {  
3      console.log(monto)  
4      monto = monto + 300  
5  }  
6
```

```
50  
350  
650  
950
```

Vemos entonces que se repiten algunas particularidades respecto a las bifurcaciones:

- La **condición** se especifica **entre paréntesis**
- Las **instrucciones** que se ejecutan **en cada iteración** del ciclo se encuentran **enmarcadas entre llaves**
- Debido a lo recién expuesto, la **tabulación** también resulta **optativa**

# Ciclos repetitivos. Do while

Muy similar a lo recién explicado es la instrucción **“do while”**. Veamos su funcionamiento con el mismo ejemplo anterior.

```
1  monto = 50;  
2  do {  
3      console.log(monto)  
4      monto = monto + 300  
5  } while (monto < 1000)  
6
```

```
50  
350  
650  
950  
1250
```

A primera vista ambos programas generan la misma salida, entonces ¿cuál es la diferencia entre ambos? Que en este último caso el ciclo iterativo se ejecuta como mínimo una vez y al finalizar el mismo se pregunta por la condición.

Por ejemplo, si el monto hubiese sido originalmente igual a 500, en el primer ejemplo el ciclo no se hubiese ejecutado ninguna vez pero en el segundo se hubiese ejecutado una vez.

# Ciclos repetitivos. For

Si en cambio conociéramos de antemano la cantidad de iteraciones, sería más adecuado utilizar un bucle **“for”**. Conozcamos su funcionamiento.

```
for (a,b,c) {  
    Instrucciones a repetir en cada iteración  
}
```

- El parámetro “a” se ejecuta siempre antes de la primer iteración
- El parámetro “b” es una condición. Si es verdadera, se realiza la próxima iteración. Si es falsa, se sale del ciclo iterativo
- El parámetro “c” se ejecuta siempre luego de cada iteración

Una vez conocida la definición formal de un bucle “for”, veamos su utilización más habitual.

```
1 ✓ for (var i=0; i<10; i++) {  
2     console.log('Iteración: ', i)  
3 }  
4
```

```
Iteración: 0  
Iteración: 1  
Iteración: 2  
Iteración: 3  
Iteración: 4  
Iteración: 5  
Iteración: 6  
Iteración: 7  
Iteración: 8  
Iteración: 9
```

Antes de seguir adelante, deberemos aprender un nuevo concepto: scope de una variable. Una definición de scope podría ser el ámbito donde dicha variable existe, y en el cual podremos acceder a su valor. El caso más común es una variable definida dentro de una función: esta variable existirá sólo dentro de la misma. Si se quisiera acceder a su valor fuera, no podríamos hacerlo.

Ahora bien, ese mismo concepto podríamos definirlo en alcances (o scopes) más pequeños. Veamos como ejemplo el siguiente bloques de código.

```
1  for (var i=0;i<4;i++) {  
2      var j = i+100;  
3      console.log('estoy dentro del scope del for')  
4  }  
5  
6  console.log(j);  
7
```

```
estoy dentro del scope del for  
estoy dentro del scope del for  
estoy dentro del scope del for  
estoy dentro del scope del for  
103
```

En este casos, podríamos definir un scope dentro de las llaves del for. Sin embargo, la variable j, definida dentro de dichas llaves, puede ser accedida fuera de dicho scope, por lo tanto, el alcance de la variable no se circunscribe exclusivamente a este scope. Veremos a continuación cómo lograr esto.



# Uso de let

En la versión ES6 de Javascript aparecieron dos palabras reservadas nuevas: **“let”** y **“const”**. Explicaremos en este slide el uso de **“let”**.

La palabra **“let”** es similar a **“var”**, pero la diferencia entre ambas es el scope en el que ambas existen. Esto último se explica a través del siguiente ejemplo.

```
1  var timer = 5
2  while (timer > 0) {
3      console.log('var:', timer)
4      if (timer >= 3) {
5          let timer = 28;
6          console.log('let:', timer)
7      }
8      timer--
9  }
10
```

```
var: 5
let: 28
var: 4
let: 28
var: 3
let: 28
var: 2
var: 1
```

La primer variable timer, que se define mediante el var, vemos que tiene vigencia en todo el programa. Sin embargo, la segunda variante timer, definida mediante let, tiene vigencia sólo en su scope. Es importante notar que dicho scope está enmarcado entre las llaves del **“if”**.

# Arrays

En forma análoga a las listas de Python, Javascript puede trabajar con **arrays**. Una array nos permite almacenar una lista de valores, cada uno de los cuales ocupa una **posición** en dicho array. Estos elementos pueden ser de cualquier tipo, y no necesariamente tienen que tener el mismo tipo. Al igual que en Python, para trabajar sobre Arrays deberemos utilizar corchetes.

```
1 let miArray = [10,"20",30,"casa",true]
2 console.log(miArray)
3 console.log(miArray[3])
4
```

```
[ 10, '20', 30, 'casa', true ]
casa
```

Como vemos en el último console.log, al utilizar el índice 3 se imprimió la 4ta posición del array. Esto nos indica que, también al igual que Python, los arrays son base 0. Es decir, su primer índice es el 0.

Existen muchísimas funciones que podemos aplicar sobre arrays, sin embargo, los iremos aprendiendo en el momento en que los necesitemos.

Si bien los objetos en Javascript tienen mayor complejidad que los diccionarios de Python, podemos comenzar a aprender sobre los mismos mediante esta aproximación. Es decir, pensar en estos objetos como una colección de claves y valores.

```
1 var persona = {  
2   nombre: 'Pedro',  
3   apellido: 'Gomez',  
4   edad: 38,  
5   mail: 'pedro.gomez@gmail.com'  
6 }  
7  
8 console.log(persona);  
9
```

```
{  
  nombre: 'Pedro',  
  apellido: 'Gomez',  
  edad: 38,  
  mail: 'pedro.gomez@gmail.com'  
}
```

A continuación algunos detalles sobre esta sintaxis:

- Al crear el objeto, todos los pares clave/valor se enmarcan entre llaves
- La asignación de clave/valor se realiza mediante clave: valor
- Cada par clave/valor se separa con una coma

Llamaremos a partir de ahora atributos a cada una de las claves. Para acceder o modificar al valor de cada uno de los atributos del objeto, podremos utilizar dos sintaxis: una mediante punto y otra mediante el uso de corchetes.

```
7  
8 console.log('El nombre es:', persona.nombre);  
9 console.log('La edad es:', persona['edad']);  
10
```

```
El nombre es: Pedro  
La edad es: 38
```

# Ciclos repetitivos. For...of y for...in

Otra forma del for es la que ya conocemos de Python, que nos permitirá iterar en los distintos elementos de un array. En este caso deberemos utilizar la sintaxis “for...of”

```
1  paises = ['Argentina', 'Chile', 'Uruguay', 'Brasil'];  
2  for (pais of paises) {  
3      console.log(pais)  
4  }  
5
```

```
Argentina  
Chile  
Uruguay  
Brasil
```

También podremos iterar en los distintos atributos de un objeto, en este caso, mediante la sintaxis “for...in”.

```
1  var auto = {  
2      marca: 'Ford',  
3      modelo: 'Fiesta',  
4      anio: 2014  
5  }  
6  
7  for (atributo in auto) {  
8      console.log(atributo, '--->', auto[atributo]);  
9  }  
10
```

```
marca ---> Ford  
modelo ---> Fiesta  
anio ---> 2014
```

# Ciclos repetitivos. Condiciones especiales

Existen algunas instrucciones que modifican el funcionamiento estándar de los ciclos iterativos. Estas son **“break”** y **“continue”**. Su funcionamiento es idéntico al de Python: la primera rompe la iteración actual y sale del ciclo iterativo, mientras que la segunda rompe la iteración actual y comienza en forma automática el próximo ciclo iterativo.

```
1  var indice = 0;
2  while (true){
3      indice ++;
4      if (indice == 5) {
5          break;
6      }
7      if (indice == 3) {
8          continue;
9      }
10     console.log('Indice:', indice);
11 }
12 console.log('Final:', indice);
13
```

```
Indice: 1
Indice: 2
Indice: 4
Final: 5
```

# Uso de const

Como se mencionó antes, una de las mejoras en la versión ES6 de Javascript fue la incorporación de la palabra reservada **“const”**. Mediante esta palabra podremos definir variables que no podrán volver a ser asignadas en ninguna otra parte del programa. Si esto ocurriese, el interprete Javascript cancelaría la ejecución mediante un error.

```
1  const PI = 3.14;  
2  console.log('PI inicializado');  
3  PI = 3.14159;  
4  console.log('PI modificado');  
5
```

```
PI inicializado  
TypeError: Assignment to constant variable.  
    at /home/runner/0ivzm1ui5srt/index.js:3:4  
    at Script.runInContext (vm.js:130:18)  
    at Object.<anonymous> (/run_dir/interp.js:209:20)
```

Es un estándar definir estas constantes en mayúscula.

Que sea constante no quiere decir que sea inmutable. Veremos un ejemplo con arrays.

```
1  const numeros = [1,2,3,4,5];  
2  console.log('Antes', numeros);  
3  numeros.push(6);  
4  console.log('Después', numeros);  
5
```

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

Esto funciona ya que no se vuelve a asignar un valor al array sino que actualiza mediante uno de sus métodos

En Javascript podemos definir funciones de varias formas. Veamos las más comunes.

```
1 function suma (n1,n2){  
2     let resultado = n1 + n2;  
3     return resultado;  
4 }  
5  
6 console.log(suma(5,3));  
7
```

```
1 let suma = function (n1,n2){  
2     let resultado = n1 + n2;  
3     return resultado;  
4 }  
5  
6 console.log(suma(5,3));  
7
```

```
1 let suma = (n1,n2) => {  
2     let resultado = n1 + n2;  
3     return resultado;  
4 }  
5  
6 console.log(suma(5,3));  
7
```

Las 3 funciones muestran por consola el mismo valor:

8

Las dos últimas formas son posibles debido a que la variable “suma” contiene una función, que como vimos al principio de la presentación es un tipo de dato no primitivo. Además, la última forma tiene dos particularidades para el caso en que la función tenga una sola línea de ejecución:

- Las llaves pasan a ser optativas
- El return queda implícito, es decir, la función devuelve el valor de la instrucción.

```
1 let suma = (n1,n2) => n1 + n2;  
2  
3 console.log(suma(5,3));  
4
```

8



# Funciones. Parámetros por valor y referencia

Al igual que en Python, los tipos de datos primitivos pasan por valor a las funciones, mientras que el resto pasa por referencia. Esto quiere decir que en el primer caso se crean copias de las variables originales, mientras que en el segundo se pasa un puntero a la misma variable.

```
1 ✓ const cambiar = (dato) => {  
2     dato = 25;  
3     console.log('En función:', dato)  
4 }  
5  
6 dato = 10;  
7 console.log('Original:', dato)  
8 cambiar(dato);  
9 console.log('Final:', dato)  
10
```

```
Original: 10  
En función: 25  
Final: 10
```

En este caso, como la variable es de un tipo de dato primitivo (numérico), la variable “dato” de la función es una copia de la variable “dato” original (ya que pasa por valor). Por lo tanto, luego de modificarse dentro, no se ve alterado su valor fuera.

```
1 ✓ const cambiar = (dato) => {  
2     dato.push(25);  
3     console.log('En función:', dato)  
4 }  
5  
6 dato = [10];  
7 console.log('Original:', dato)  
8 cambiar(dato);  
9 console.log('Final:', dato)  
10
```

```
Original: [ 10 ]  
En función: [ 10, 25 ]  
Final: [ 10, 25 ]
```

Aquí, como la variable es de un tipo de dato no primitivo (array), la variable “dato” de la función apunta a la misma variable “dato” original (ya que pasa por referencia). Por lo tanto, luego de modificarse dentro, se ve alterado su valor fuera.

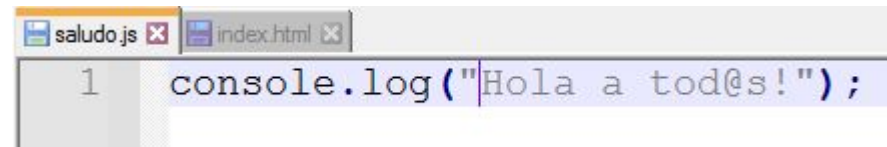


# Ejecución de un programa JS desde HTML

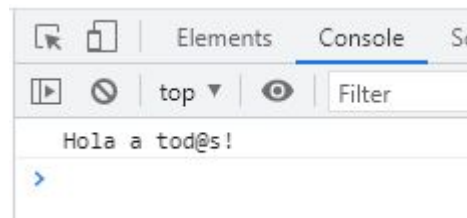
Ahora que ya conocemos las bases de Javascript, deberemos aprender cómo se ejecuta un programa desde una página HTML. Para ello, deberemos utilizar la etiqueta **<script>**. Su uso más básico es el siguiente:

```
<script src=nombre-del-archivo></script>
```

Por ejemplo, mediante `<script src="saludo.js"></script>` estaremos ejecutando el programa saludo.js. Una buena práctica implica incorporar la etiqueta script al final del body, ya conoceremos más adelante el motivo. Veamos entonces qué contiene saludo.js:

A screenshot of a code editor with two tabs: 'saludo.js' and 'index.html'. The 'saludo.js' tab is active, showing a single line of JavaScript code: `1 console.log("Hola a tod@s!");`. The line number '1' is visible in the left margin.

En este caso, por tanto, deberemos ver en la consola el saludo impreso en este programa:

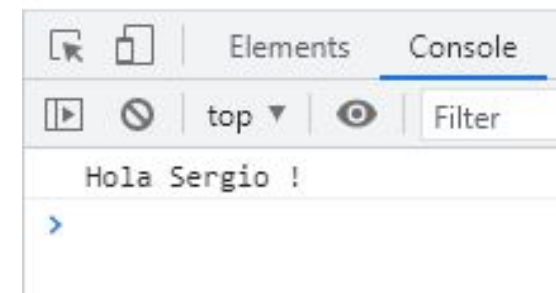
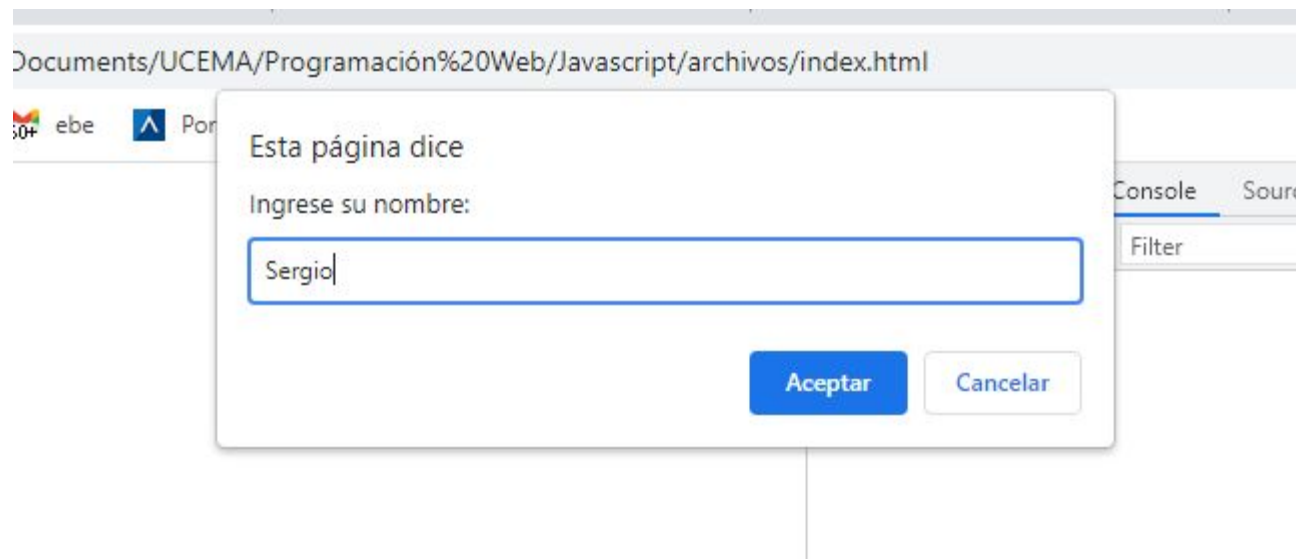
A screenshot of a browser's developer console. The 'Console' tab is selected, showing a single log entry: 'Hola a tod@s!'. Above the log entry, there are icons for 'Elements' and 'Console', and a 'Filter' input field. A blue prompt character '>' is visible at the bottom of the console.

# Ingreso de datos

Si bien lo más normal es que el usuario ingrese los datos a través de etiquetas `<input>`, existe otra forma de realizarlo a través de una instrucción Javascript, la cual puede ser de utilidad para hacer pruebas rápidas. Dicha instrucción es **“prompt”**.

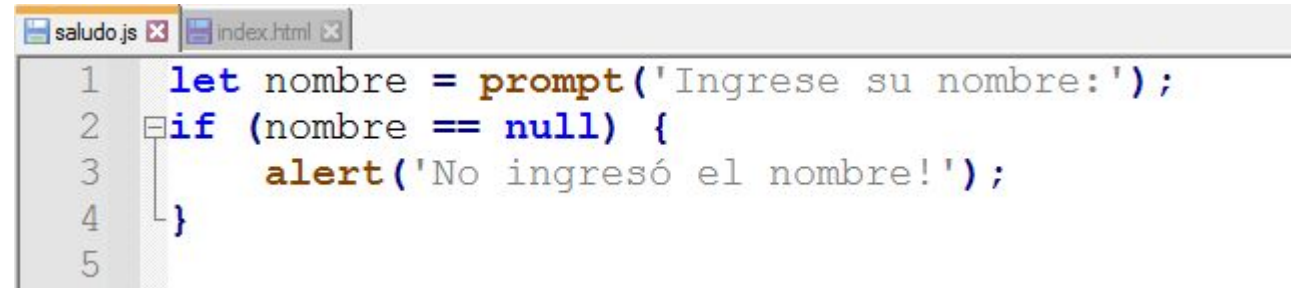
```
saludo.js x index.html x
1 let nombre = prompt('Ingrese su nombre:');
2 console.log('Hola', nombre, '!');
3
```

Cabe destacar que si se presiona “Cancelar”, la función prompt devuelve el valor **null**.

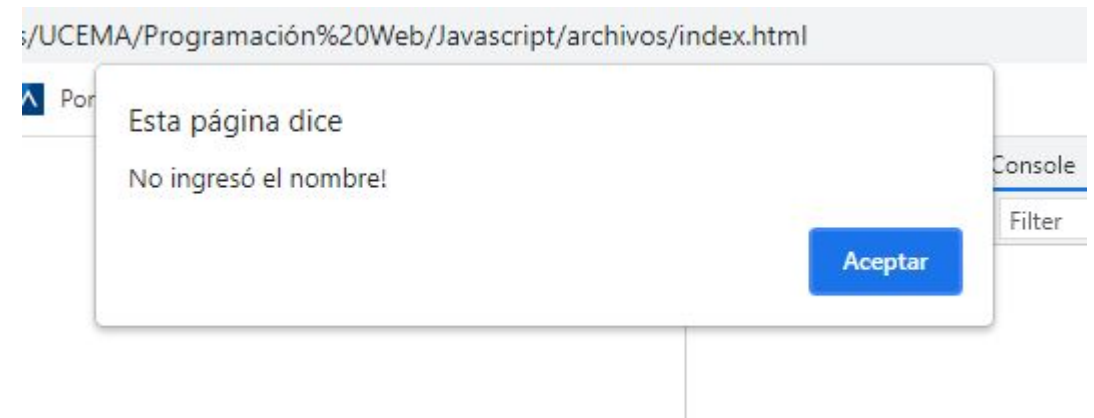
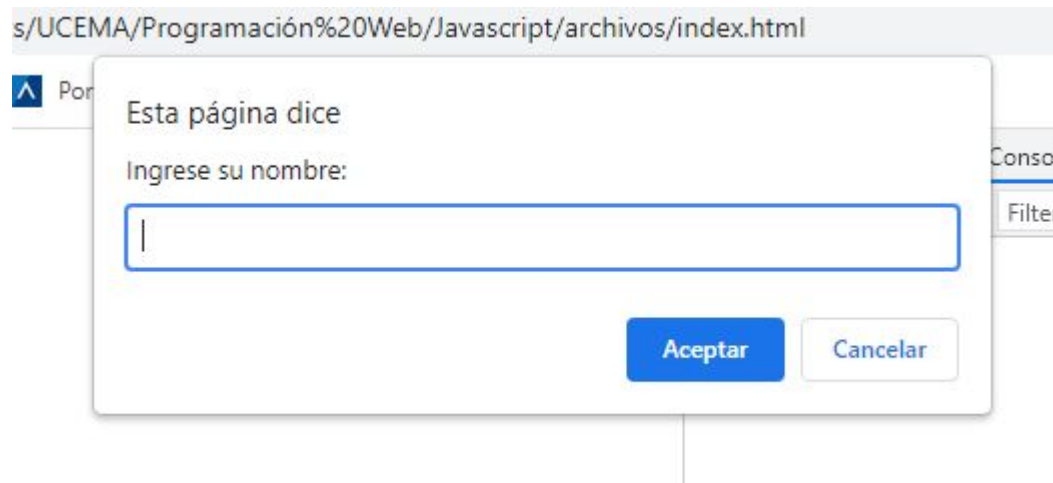


# Mensajes al usuario

De la misma forma que se comentó recién, lo más normal para enviar mensajes al usuario será a través de elementos de la página. Sin embargo, una forma sencilla de enviar mensajes al usuario es mediante la instrucción **“alert”**.



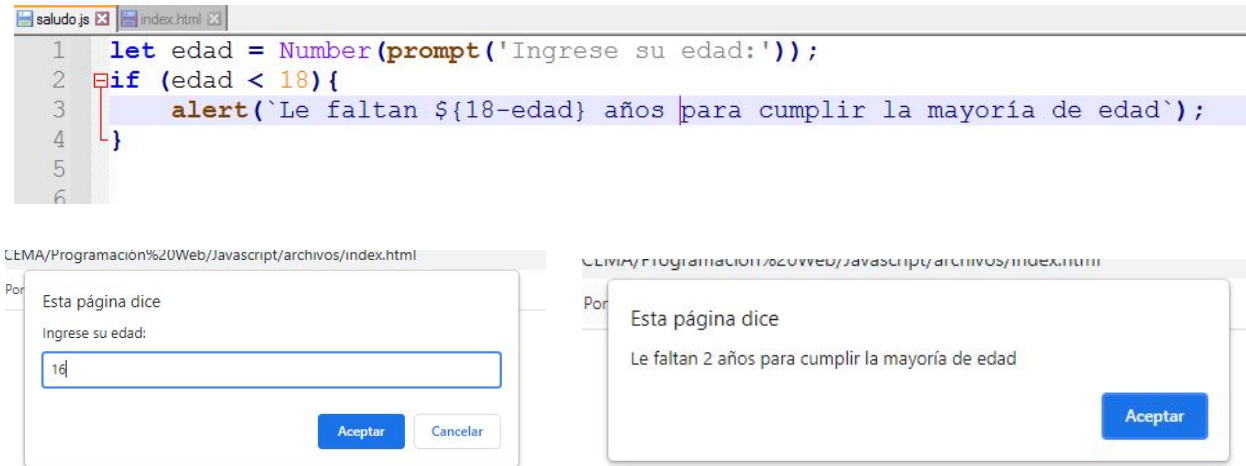
```
saludo.js | index.html
1  let nombre = prompt('Ingrese su nombre:');
2  if (nombre == null) {
3      alert('No ingresó el nombre!');
4  }
5
```



# Conversión de tipo de datos

Dado que Javascript, a diferencia de Python, es un lenguaje débilmente tipado, normalmente no es necesario realizar conversión de tipos de datos, ya que es el mismo interprete quien se ocupa de esta tarea cuando la considera necesaria. Sin embargo, si queremos forzar la conversión podremos realizarlo mediante las siguientes funciones:

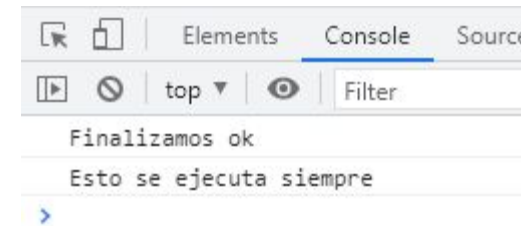
Función	Convierte a...
Number(x)	Number
parseFloat(x)	Number con decimales
parseInt(x)	Number entero
String(n)	String



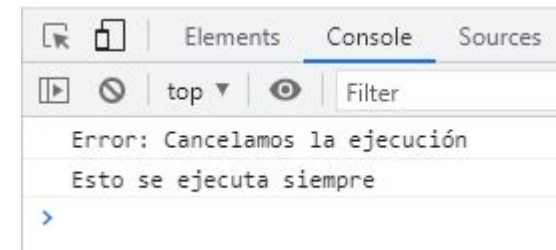
# Excepciones

Al igual que en Python, podremos controlar errores inesperados que puedan suceder en un programa, así como también lanzar excepciones expreso. Veamos su funcionamiento completo a través del siguiente ejemplo.

```
saludo.js x index.html x
1 let cancelar = prompt('Ir al catch (si/no):');
2 try{
3   if (cancelar == 'si'){
4     throw 'Cancelamos la ejecución';
5   }
6   console.log('Finalizamos ok');
7 } catch (e){
8   console.log('Error:', e);
9 } finally {
10   console.log('Esto se ejecuta siempre');
11 }
12
```



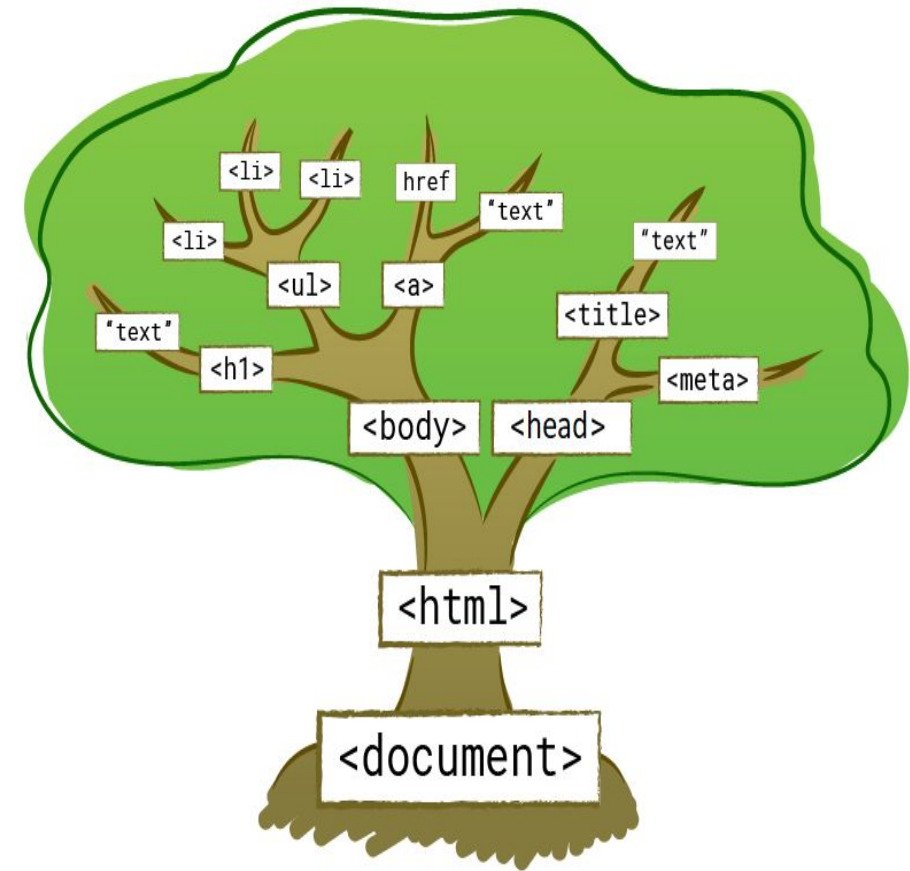
Si se ingresa un valor distinto a "si".



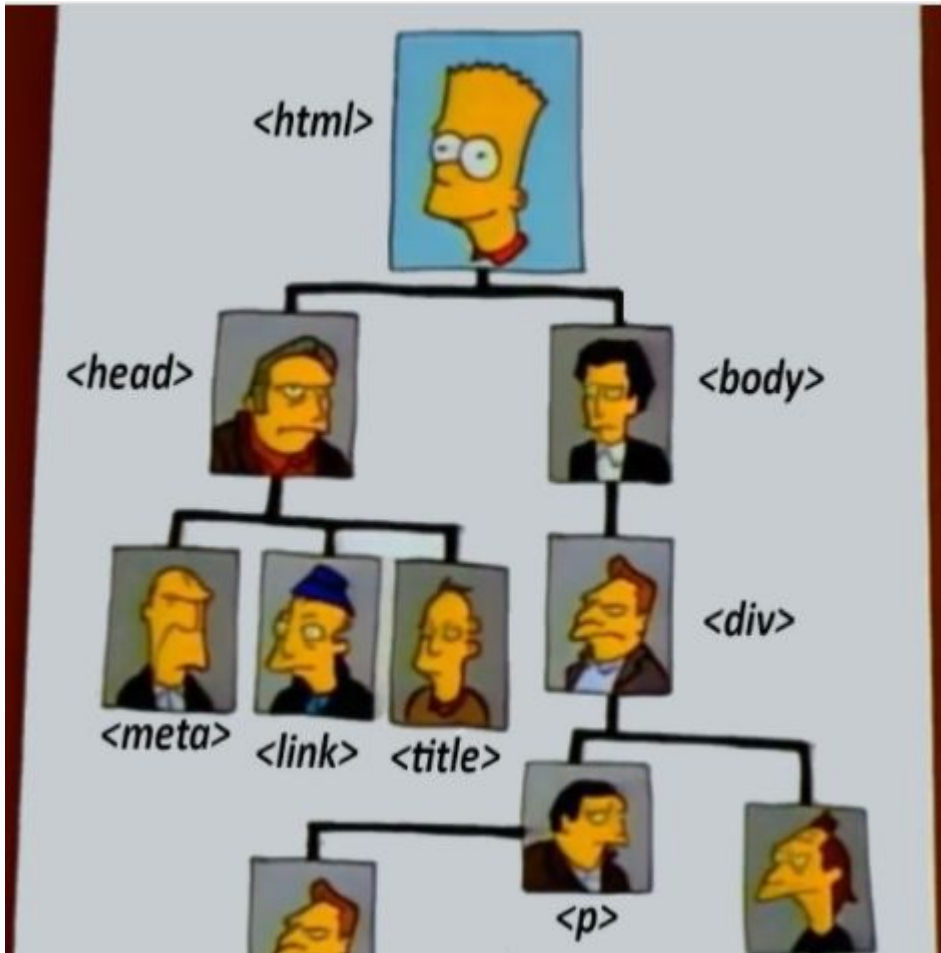
Si se ingresa el valor "si".

## ¿Y el dinamismo? Conozcamos al DOM...

Ahora que ya sabemos los fundamentos de Javascript, es hora de darle a las páginas el prometido dinamismo. Para ello, deberemos comenzar descubriendo el concepto de DOM y cómo es posible manipularlo para alterar en forma dinámica las páginas. De esta forma, podremos incorporar nueva información, alterar la ya existente o bien eliminar parte de la misma.



# DOM. Document Object Model



El Modelo de Objetos del Documento (DOM) es una estructura de objetos generada por el navegador, la cual representa la página HTML actual. Con JavaScript la empleamos para acceder y modificar de forma dinámica elementos de la interfaz.

Es decir que, por ejemplo, desde Javascript podemos modificar el texto contenido de una etiqueta `<h1>`.



# DOM. ¿Cómo funciona?

La estructura de un documento HTML son las etiquetas. En el Modelo de Objetos del Documento (DOM), cada etiqueta HTML es un objeto, al que podemos llamar nodo. Las etiquetas anidadas son llamadas “nodos hijos” de la etiqueta “nodo padre” que las contiene.

Todos estos objetos son accesibles empleando JavaScript mediante el objeto global document. Por ejemplo, document.body es el nodo que representa la etiqueta <body>.

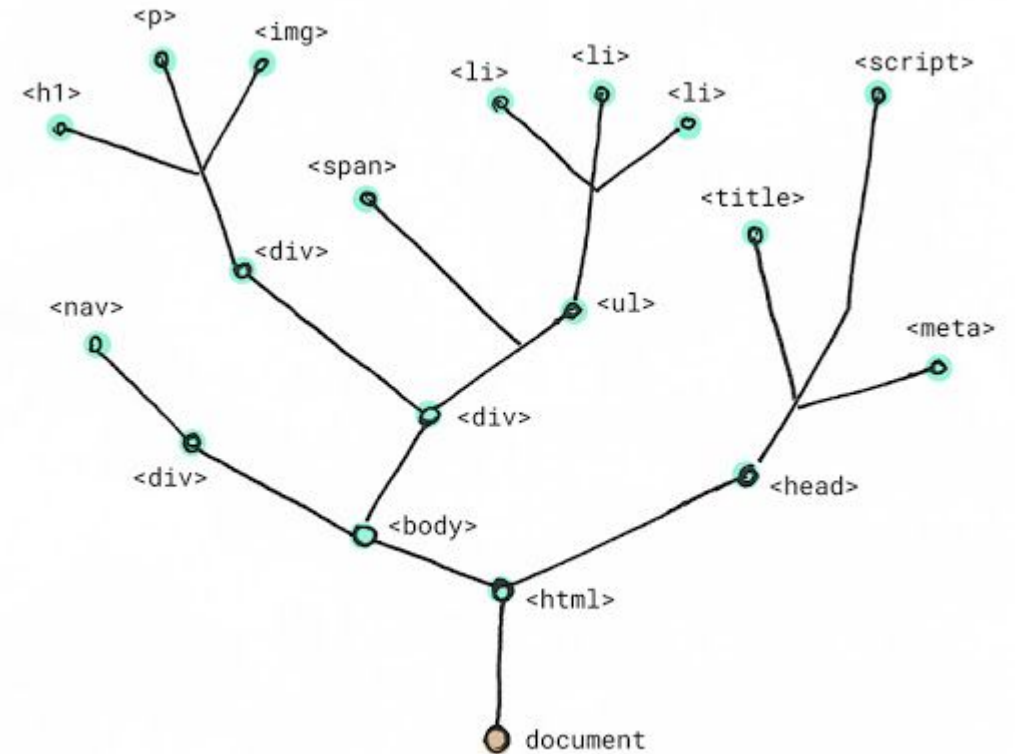
```
1 <!DOCTYPE html>
2 <head>
3   <title> Mi primer App </title>
4 </head>
5 <body>
6   <h1> Listado de Clientes </h1>
7 </body>
8 </html>
9
```

```
DOCTYPE: html
HTML
  HEAD
    #text:
    TITLE
      #text: Mi primer App
    #text:
  #text:
  BODY
    #text:
    H1
      #text: Listado de Clientes
    #text:
```



# Estructura del DOM

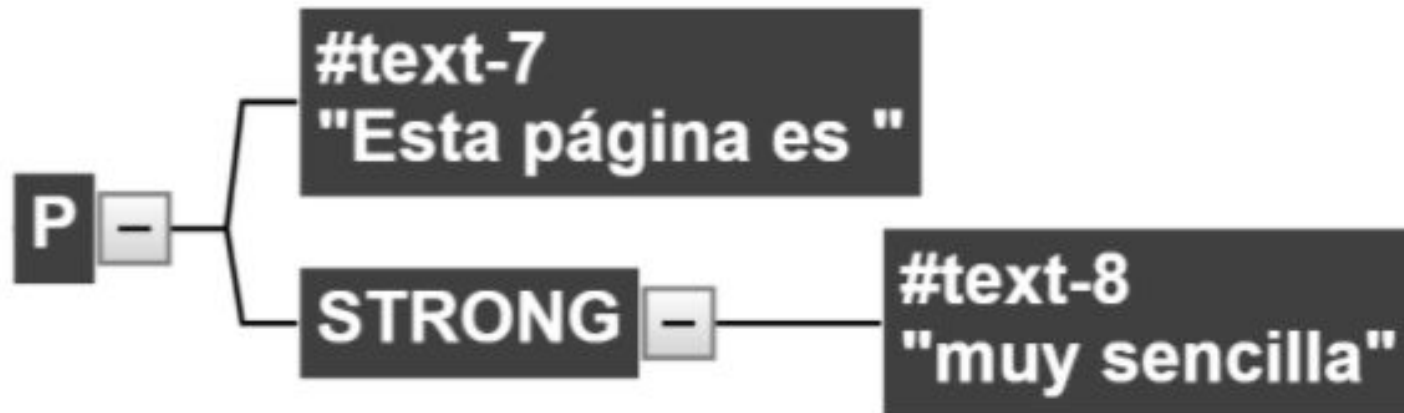
- ✓ Cada etiqueta HTML se transforma en un nodo de tipo "Elemento". La conversión de etiquetas en nodos se realiza de forma jerárquica.
- ✓ De esta forma, del nodo raíz solamente pueden derivar los nodos HEAD y BODY.
- ✓ A partir de esta derivación inicial, cada etiqueta HTML se transforma en un nodo que deriva del correspondiente a su "etiqueta padre".
- ✓ La transformación de las etiquetas HTML habituales genera dos nodos: el primero es el nodo de tipo "Elemento" (correspondiente a la propia etiqueta HTML) y el segundo es un nodo de tipo "Texto" que contiene el texto encerrado por esa etiqueta HTML.



# Ejemplo de un nodo

```
<p>Esta página es <strong>muy sencilla</strong></p>
```

La etiqueta <p> se transforma en los siguientes nodos del DOM:

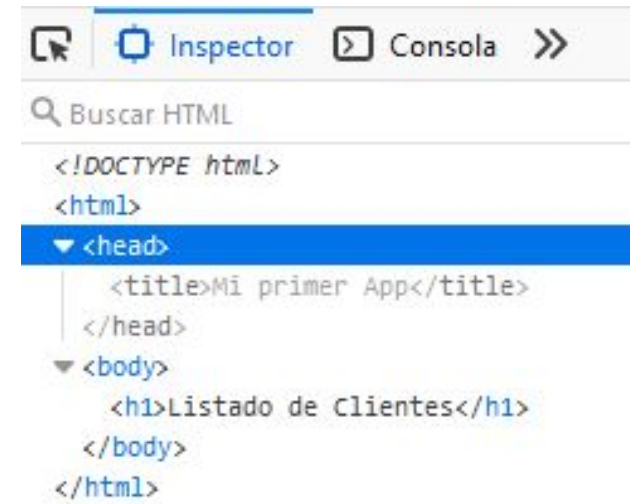


# Visualizando el DOM desde el navegador

Los navegadores modernos brindan medios para editar el DOM de cualquier página en tiempo real.

Por ejemplo, en Chrome, podemos hacerlo mediante la Herramienta para desarrolladores en la pestaña “Elements”.

Si bien la estructura DOM está simplificada, es un medio muy útil para verificar y probar en tiempo real actualizaciones en la estructura.



# Tipos de nodos

---

La especificación completa de DOM define 12 tipos de nodos, aunque los más usados son:

- Document, nodo raíz del que derivan todos los demás nodos del árbol.
- Element, representa cada una de las etiquetas HTML. Se trata del único nodo que puede contener atributos y el único del que pueden derivar otros nodos.
- Attr, se define un nodo de este tipo para representar cada uno de los atributos de las etiquetas HTML, es decir, uno por cada par atributo=valor.
- Text, nodo que contiene el texto encerrado por una etiqueta HTML.
- Comment, representa los comentarios incluidos en la página HTML.

# Acceso a los nodos

Hemos aprendido ya qué son los nodos. Lo siguiente es cómo podemos acceder a los mismos y alterarlos para modificar de esta forma el contenido de la página de manera dinámica. Conozcamos entonces las tres principales formas de acceder a un nodo.

- `getElementById()`: obtiene un único nodo a partir del id del mismo
- `getElementsByClassName()`: obtiene un array de nodos cuya clase sea la que enviamos por parámetro
- `getElementsByTagName()`: obtiene un array de nodos cuya etiqueta sea del tipo que enviamos por parámetro

En los siguientes slides veremos a detalle cada uno, para lo cual nos basaremos en la página que se muestra a la derecha.

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="utf-8">
    <title>Practicando con el DOM</title>
  </head>
  <body>
    <div id="app">
      <div id="parrafo1"> Hola a tod@s! </div>
    </div>
    <ul>
      <li class="pais"> AR </li>
      <li class="pais"> UY </li>
      <li class="pais"> CL </li>
    </ul>
    <div>
      <div>CONTENEDOR 1</div>
      <div>CONTENEDOR 2</div>
    </DIV>
    <script src="accesodom.js"></script>
  </body>
</html>
```

# getElementById. Obtener un nodo por su id

```
let div = document.getElementById("app");  
let parrafo = document.getElementById("parrafo1");  
console.log('Contenido div: ', div.innerHTML);  
console.log('Contenido párrafo: ', parrafo.innerHTML);
```

```
Contenido div:  
<div id="parrafo1"> Hola a tod@s! </div>
```

```
Contenido párrafo:  Hola a tod@s!
```

Veamos paso a paso lo que hemos hecho. Mediante la primer instrucción, obtenemos el nodo cuyo id es igual a “app” (es importante destacar que, debido a su naturaleza, el método getElementById devuelve como máximo un único nodo). En este caso, el nodo que se recupera es el siguiente:

```
<div id="app">  
  <div id="parrafo1"> Hola a tod@s! </div>  
</div>
```

Luego, hacemos lo propio con el nodo cuyo id es igual a “parrafo 1”, obteniéndose en este caso:

```
<div id="parrafo1"> Hola a tod@s! </div>
```

Finalmente, con las dos últimas instrucciones se accede al atributo innerHTML de ambos nodos. Este atributo es el texto que se encuentra en dicho nodo, por ello en la consola se visualizan dichos textos.

# getElementsByClassName. Obtener nodos por su clase

```
let paises = document.getElementsByClassName("pais");  
console.log('pais 1: ', paises[0].innerHTML);  
console.log('pais 2: ', paises[1].innerHTML);  
console.log('pais 3: ', paises[2].innerHTML);
```

pais 1:	AR
pais 2:	UY
pais 3:	CL

Siguiendo la misma metodología del slide anterior, veamos qué nodos obtenemos luego de la primer instrucción.

```
<li class="pais"> AR </li>  
<li class="pais"> UY </li>  
<li class="pais"> CL </li>
```

En este caso, debido a su naturaleza, el método `getElementsByClassName` devuelve siempre un array con los nodos obtenidos, incluso en aquel caso donde sólo se obtenga un único nodo.

Luego, con las tres últimas instrucciones se accede al atributo `innerHTML` de los nodos recuperados. Al ser un array accedemos a la posición 0, 1 y 2 del mismo. Como cada uno de esos elementos es un nodo, es posible acceder al atributo `innerHTML` de cada uno de ellos.



# getElementsByTagName. Otra forma de iterar

Como lo más común es que no conozcamos de antemano la cantidad de nodos que vamos a recuperar, la forma más común para iterar será la siguiente:

```
for (pais of paises) {  
    console.log('pais: ', pais.innerHTML);  
}
```

pais:	AR
pais:	UY
pais:	CL

De esta forma, pais será una variable que contendrá en cada una de las iteraciones un nodo distinto. La primera vez, entonces, se recuperará el siguiente nodo:

```
<li class="pais"> AR </li>
```

La segunda:

```
<li class="pais"> UY </li>
```

Y la tercera:

```
<li class="pais"> CL </li>
```



# getElementsByTagName. Obtener nodos por su tipo

```
let contenedores = document.getElementsByTagName("div");  
for (contenedor of contenedores) {  
    console.log('contenedor: ', contenedor.innerHTML);  
}
```

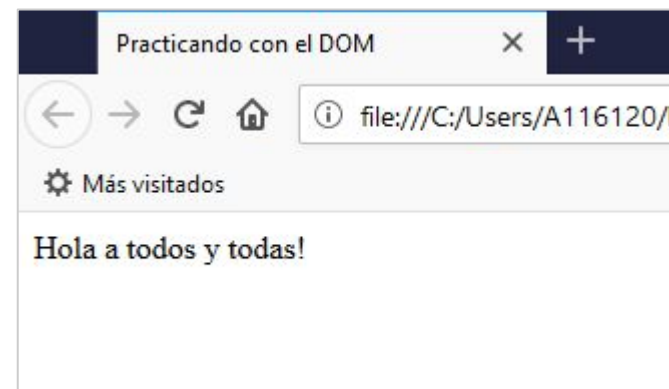
```
contenedor:  
  <div id="parrafo1"> Hola a tod@s! </div>  
  
contenedor:  Hola a tod@s!  
contenedor:  
  <div>CONTENEDOR 1</div>  
  <div>CONTENEDOR 2</div>  
  
contenedor:  CONTENEDOR 1  
contenedor:  CONTENEDOR 2
```

Este caso es similar a `getElementsByClassName` donde recuperamos los nodos en un array, ya que debido a la naturaleza de `getElementsByTagName`, este método podrá recuperar más de un nodo. En este ejemplo, se recuperan todos los nodos de tipo “div” y luego se itera en los mismos imprimiendo por consola su contenido `innerHTML`.

# Modificación de nodos en forma dinámica

Ahora que ya sabemos recuperar un nodo, podremos modificar en forma muy sencilla su contenido. Veamos un ejemplo.

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="utf-8">
    <title>Practicando con el DOM</title>
  </head>
  <body>
    <div id="app">
      <div id="parrafo1"> Hola a tod@s! </div>
    </div>
    <script src="modifdom.js"></script>
  </body>
</html>
```



```
let parrafo = document.getElementById("parrafo1");
parrafo.innerHTML = "Hola a todos y todas!";
```

# Creación de elementos. Inserción de nodos en el DOM

Para insertar contenido nuevo en el DOM deberemos primero crear el nodo, en este caso uno de tipo “element”. Luego, con el nodo ya creado, deberemos incorporarlo en el lugar de DOM que deseemos. Empecemos, por tanto, creando el elemento.

```
let nuevoParrafo = document.createElement("p");  
nuevoParrafo.innerHTML = "<h2>¡Estoy apareciendo en forma dinámica!</h2>";
```

En la variable nuevoParrafo se almacena el nodo que queremos insertar. Comencemos insertándolo directamente en el body del HTML y veamos cómo queda configurado el DOM.

```
document.body.appendChild(nuevoParrafo);
```

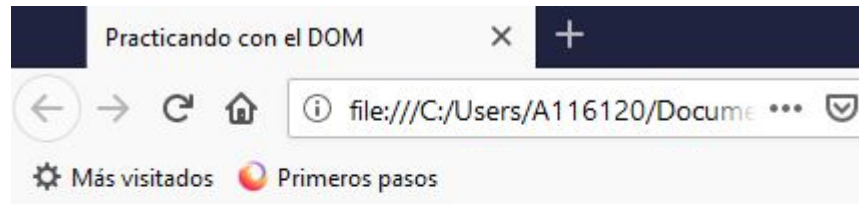


```
<!DOCTYPE html>  
<html lang="es">  
  <head> ... </head>  
  <body>  
    <div id="app">  
      <div id="parrafo1">Hola a todos y todas!</div>  
    </div>  
    <script src="modifdom.js"></script>  
    <p>  
      <h2>¡Estoy apareciendo en forma dinámica!</h2>  
    </p>  
  </body>  
</html>
```

# Creación de elementos. Inserción de nodos en el DOM

Ahora bien, si hubiésemos querido incorporar el nodo en un nodo inferior a body, ¿cómo tendríamos que haber hecho? En este caso, primero deberemos recuperar el nodo padre y luego insertar el nuevo elemento como su hijo. Veámoslo con el siguiente ejemplo: si lo hubiésemos querido “colgar” del div cuyo id es app.

```
let parrafoApp = document.getElementById("app");  
parrafoApp.appendChild(nuevoParrafo);
```



Hola a todos y todas!

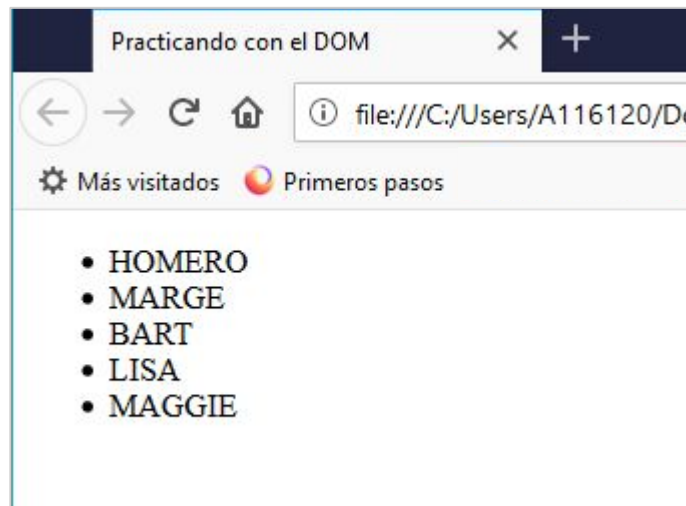
**¡Estoy apareciendo en forma dinámica!**

```
<!DOCTYPE html>  
<html lang="es">  
  <head> ... </head>  
  <body>  
    <div id="app">  
      <div id="parrafo1">Hola a todos y todas!</div>  
      <p>  
        <h2>¡Estoy apareciendo en forma dinámica!</h2>  
      </p>  
    </div>  
    <script src="modifdom.js"></script>  
  </body>  
</html>
```

# Inserción de nodos en el DOM desde un array

Veamos un simple ejemplo donde insertaremos varios elementos en una lista a partir de un array.

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="utf-8">
    <title>Practicando con el DOM</title>
  </head>
  <body>
    <ul id="personas">
    </ul>
    <script src="modifdom.js"></script>
  </body>
</html>
```



```
let padre = document.getElementById("personas");
let personas = ["HOMERO", "MARGE", "BART", "LISA", "MAGGIE"];
for (const persona of personas) {
  let li = document.createElement("li");
  li.innerHTML = persona;
  padre.appendChild(li);
}
```

```
<!DOCTYPE html>
<html lang="es">
  <head> </head>
  <body>
    <ul id="personas">
      <li>
        ::marker
        HOMERO
      </li>
      <li> </li>
      <li> </li>
      <li> </li>
      <li> </li>
    </ul>
    <script src="modifdom.js"></script>
  </body>
</html>
```



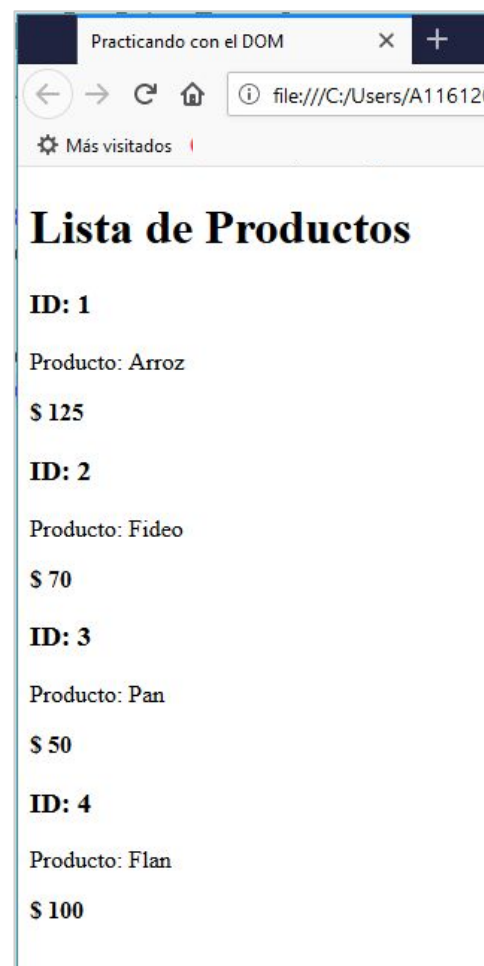
# Inserción de nodos utilizando template strings

El uso de template strings es especialmente útil al insertar nodos en una página HTML. Veamos un ejemplo.

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="utf-8">
    <title>Practicando con el DOM</title>
  </head>
  <body>
    <h1> Lista de Productos </h1>
    <script src="modifdom.js"></script>
  </body>
</html>
```

```
const productos = [{ id: 1, nombre: "Arroz", precio: 125 },
  { id: 2, nombre: "Fideo", precio: 70 },
  { id: 3, nombre: "Pan", precio: 50 },
  { id: 4, nombre: "Flan", precio: 100 }];

for (const producto of productos) {
  let contenedor = document.createElement("div");
  contenedor.innerHTML = `<h3> ID: ${producto.id}</h3>
    <p> Producto: ${producto.nombre}</p>
    <b> $ ${producto.precio}</b>`;
  document.body.appendChild(contenedor);
}
```



```
<!DOCTYPE html>
<html lang="es">
  <head> ... </head>
  <body>
    <h1>Lista de Productos</h1>
    <script src="modifdom.js"></script>
    <div>
      <h3>ID: 1</h3>
      <p>Producto: Arroz</p>
      <b>$ 125</b>
    </div>
    <div> ... </div>
    <div> ... </div>
    <div> ... </div>
  </body>
</html>
```

# Eliminación de nodos

No existe una forma directa de eliminar el nodo. El procedimiento es recuperar un nodo, acceder a su padre y desde allí eliminar el nodo hijo. El procedimiento es el siguiente:

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="utf-8">
    <title>Practicando con el DOM</title>
  </head>
  <body>
    <h1>Ejemplo de borrado de nodos </h1>
    <p id="parrafo1">
      Este es el primer párrafo
    </p>
    <p id="parrafo2">
      Este es el segundo párrafo
    </p>
    <script src="modifdom.js"></script>
  </body>
</html>
```

```
let parrafo = document.getElementById("parrafo1");
parrafo.parentNode.removeChild(parrafo);
```



```
<!DOCTYPE html>
<html lang="es">
  <head>
  </head>
  <body>
    <h1>Ejemplo de borrado de nodos</h1>
    <p id="parrafo2">Este es el segundo párrafo</p>
    <script src="modifdom.js"></script>
  </body>
</html>
```

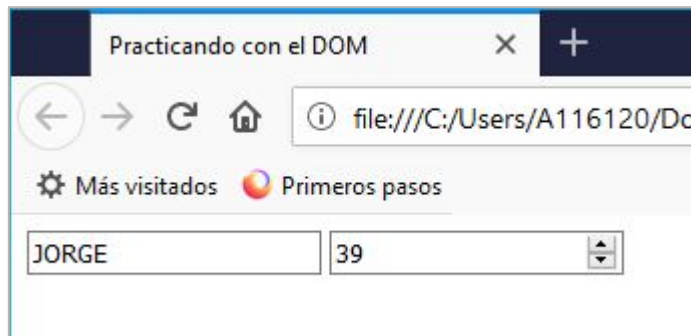


# Acceso a valores del input

Mediante el DOM, podremos acceder a los valores ingresados en los distintos input, tanto sea para consultarlos como para alterarlos. Veamos un ejemplo donde ingresamos valores directamente desde Javascript.

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="utf-8">
    <title>Practicando con el DOM</title>
  </head>
  <body>
    <form>
      <input id = "nombre" type="text">
      <input id = "edad" type="number">
    </form>
    <script src="modifdom.js"></script>
  </body>
</html>
```

```
document.getElementById("nombre").value = "JORGE";
document.getElementById("edad").value = 39;
```



# ¿Qué me está pasando?

---

A esta altura, las páginas web ya tienen vida propia, pero nos falta algo muy importante para interactuar con ellas. Por ejemplo, ¿cómo saben que hicimos click sobre un botón? ¿O qué hemos modificado un dato? ¿Y si quisiéramos ejecutar alguna validación cuando un dato pierde el foco? ¿O si nos importara saber cuando se presionó una tecla en algún formulario? Para poder detectar cada una de estas situaciones, y muchas más, deberemos conocer los eventos de Javascript y cómo manejarlos.



# ¿Qué son los eventos?



Los eventos son la manera que tenemos en Javascript de controlar las acciones de los usuarios, y definir un comportamiento de la página o aplicación en el momento en que suceden.

Con Javascript podemos definir qué es lo que debe ocurrir cuando se produce un evento. Por ejemplo, cuando el usuario hace click en cierto elemento (como un botón), cuando escribe en un campo, etc.

JavaScript permite asignar una función a cada uno de los eventos. De esta forma, cuando se produce cualquier evento, JavaScript ejecuta su función asociada. Este tipo de funciones se denominan *event handlers* ("manejadores de eventos").

Los eventos se asocian a cada elemento al cual se lo quiere "escuchar".

# Definición de un evento. Opción 1

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mi primer App</title>
  </head>
  <body>
    <button id="btnPrincipal">CLICK</button>
    <script>
      let boton = document.getElementById("btnPrincipal")
      boton.addEventListener("click", respuestaClick)
      function respuestaClick(){
        console.log("Respuesta evento");
      }
    </script>
  </body>
</html>
```

El método `addEventListener()` permite definir qué evento escuchar sobre cualquier elemento del HTML. En este caso vemos que aplica al botón ya que es el nodo que se seleccionó previamente.

El primer parámetro del `addEventListener` corresponde al nombre del evento (en este caso el evento “click”) y el segundo a la función de respuesta.

En síntesis, lo que hace este código es: cuando se hace “click” sobre el botón, se ejecuta la función `respuestaClick`.

# Definición de un evento. Opción 2

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mi primer App</title>
  </head>
  <body>
    <button id="btnPrincipal">CLICK</button>
    <script>
      let boton = document.getElementById("btnPrincipal")
      boton.onclick = () =>{console.log("Respuesta 2")}
    </script>
  </body>
</html>
```

Otra opción es emplear una propiedad del nodo para definir la respuesta al evento. Las propiedades se identifican con el nombre del evento y el prefijo *on*.

En este caso, armamos un handler para el evento click del botón, y ejecutamos una función anónima cuando se presente dicho evento.

El punto débil de esta forma es que, para definir el evento, no puede utilizarse el contenido de una variable.

# Definición de un evento. Opción 3

---

```
<input type="button" value="CLICK2" onclick="alert('Respuesta 3');" />
```

La última opción implica especificar el manejador de evento en el atributo de una etiqueta HTML. La denominación del atributo es idéntica al de la propiedad de la opción 2 (prefijo *on*).

La función puede declararse entre la comillas o bien hacer referencia a una función del script. En términos generales, es código Javascript válido.

Esta opción **no es recomendada**, aunque puede llegar a ser útil para hacer pruebas rápidas.



# Eventos del mouse

Sería imposible listar todos los eventos existentes. Para una completa referencia, se puede ingresar a [https://developer.mozilla.org/es/docs/Web/Events#eventos\\_est%C3%A1ndar](https://developer.mozilla.org/es/docs/Web/Events#eventos_est%C3%A1ndar).

De todas formas, veremos a continuación algunos eventos que es probable que tengamos que utilizar. Comencemos con los eventos relacionados al mouse. Entre ellos, se destacan los siguientes:

- **mousedown/mouseup**: se oprime/suelta el botón del ratón sobre un elemento.
- **mouseover/mouseout**: el puntero del mouse se mueve sobre/sale del elemento.
- **mousemove**: el movimiento del mouse sobre el elemento activa el evento.
- **click**: se activa después de mousedown o mouseup sobre un elemento válido.

```
//CODIGO HTML DE REFERENCIA
<button id="btnMain">CLICK</button>

//CODIGO JS
let boton      = document.getElementById("btnMain");
boton.onclick  = () => {console.log("Click")};
boton.onmousemove = () => {console.log("Move")}
```



# Eventos de teclado

Los eventos de teclado o KeyboardEvent describen una interacción del usuario con el teclado. Entre ellos se destacan:

- **keydown:** Cuando se presiona una tecla.
- **keyup:** Cuando se suelta una tecla.

```
//CODIGO HTML DE REFERENCIA
<input id = "nombre" type="text">
<input id = "edad" type="number">
//CODIGO JS
let input1 = document.getElementById("nombre");
let input2 = document.getElementById("edad");
input1.onkeyup = () => {console.log("keyUp")};
input2.onkeydown = () => {console.log("keyDown")};
```

# Evento Change

El evento change se activa cuando se detecta un cambio en el valor del elemento y el mismo pierde el foco. Por ejemplo, mientras estamos escribiendo en un input de tipo texto, no se dispara el evento change. Sin embargo, pero cuando dicho input pierde el foco, sólo cuando se detecta que fue modificado su valor, ocurre el evento *change*.

```
//CODIGO HTML DE REFERENCIA
<input id = "nombre" type="text">
<input id = "edad" type="number">

//CODIGO JS
let input1 = document.getElementById("nombre");
let input2 = document.getElementById("edad");
input1.onChange = () => {console.log("valor1")};
input2.onChange = () => {console.log("valor2")};
```

# Evento Submit

El evento submit se activa cuando el formulario es enviado. Normalmente es utilizado para validar el formulario previo a su envío al servidor o bien para abortar el envío y procesarlo con JavaScript. En este último caso, se utiliza el método preventDefault() que previene el comportamiento por default que tiene el formulario impidiendo el viaje al servidor.

```
//CODIGO HTML DE REFERENCIA
<form id="formulario">
  <input type="text">
  <input type="number">
  <input type="submit" value="Enviar">
</form>

//CODIGO JS
let miFormulario = document.getElementById("formulario");
miFormulario.addEventListener("submit", validarFormulario);

function validarFormulario(e) {
  e.preventDefault();
  console.log("Formulario Enviado");
}
```

**Muchas Gracias!!**

UNIVERSIDAD DEL CEMA  
**UCEMA**

[ucema.edu.ar](http://ucema.edu.ar)