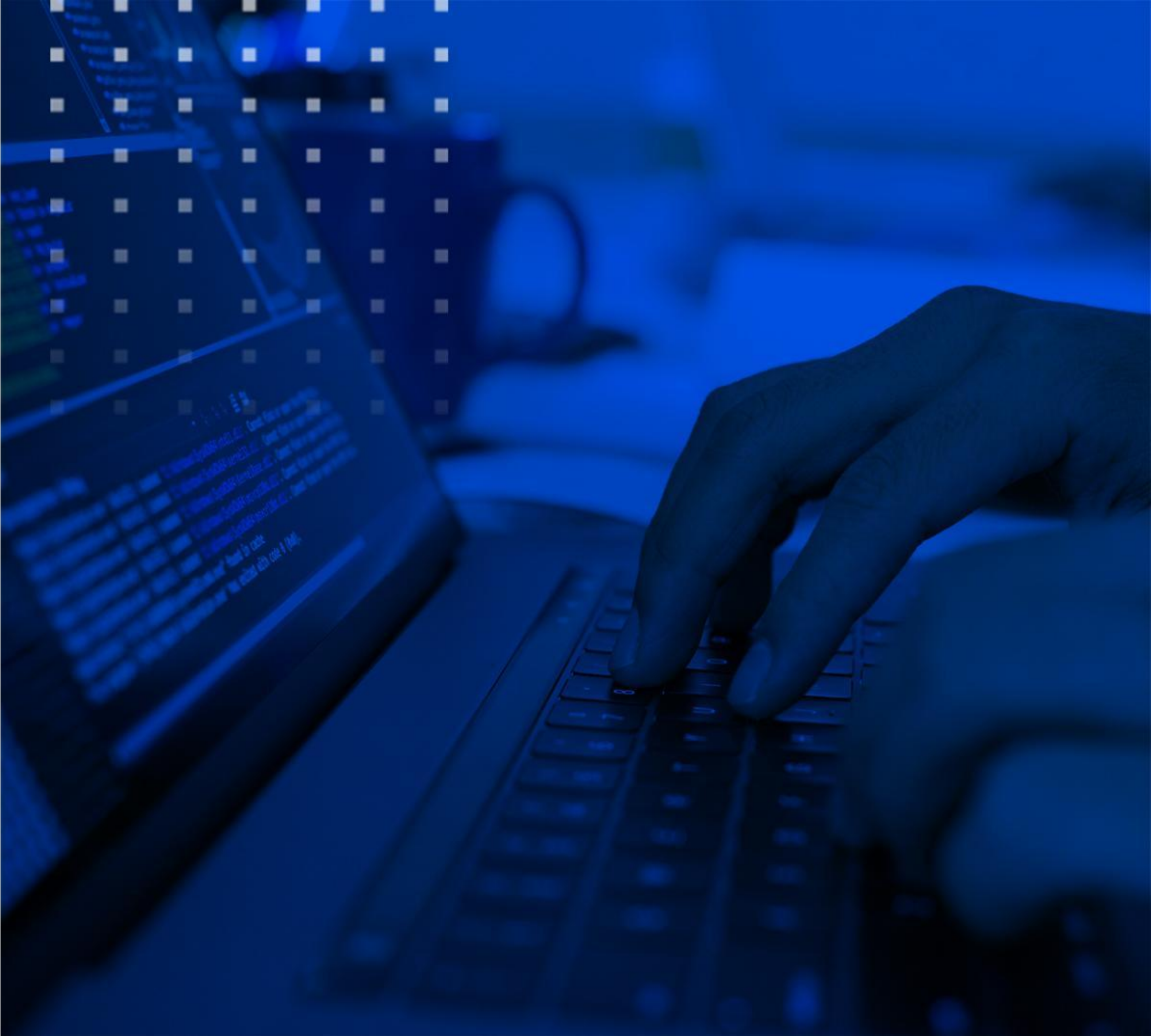
[i]ncluyeme.com

UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

# CURSO

## FULL STACK DEVELOPER
## NIVEL INICIAL

### UNIDAD 8
JavaScript: Objetos y manipulación del DOM



# Full Stack Developer Inicial

## Objetos: Conceptos generales

# ¿Qué es un objeto?

En programación, y también en JS, un objeto es una colección de datos relacionados con funcionalidad, que generalmente consta de variables, denominadas propiedades, y funciones asociadas, llamadas métodos.

Por ejemplo: el objeto *persona*, tendría como una propiedad la altura, y como un método, hablar.

# Objeto vs arrays

## Objeto Literal

```
let persona = {
    nombre: "Leia",
    apellido: "Organa",
    edad: 32
}
```

## Array

```
let nombres = [
    "Maria",
    "Sofia",
    "Sheila"
]
```

# Objeto vs arrays

**Objeto Literal**

```
let persona = {
    nombre: "Leia",
    apellido: "Organa",
    edad: 32
}
```

**Array**

```
let nombres = [
    "Maria",
    "Sofia",
    "Sheila"
]
```

**#1 -** Los objetos literales encierran sus características entre llaves {} mientras que los arrays lo hacen entre corchetes []

# Objeto vs arrays

**Objeto Literal**

```
let persona = {
    nombre: "Leia",
    apellido: "Organa",
    edad: 32
}
```

**Array**

```
let nombres = [
    "Maria",
    "Sofia",
    "Sheila"
]
```

**#2 -** En un array cada elemento se identifica mediante su posición. En un objeto mediante su **nombre de atributo**

# Objeto vs arrays

**Objeto Literal**

```
let persona = {
    nombre: "Leia",
    apellido: "Organa",
    edad: 32
}
```

**Array**

```
let nombres = [
    "Maria",
    "Sofia",
    "Sheila"
]
```

**#3** - Dado que en los objetos cada atributo tiene un nombre, usamos los dos puntos para separar el nombre del valor

# Objeto vs arrays

**Objeto Literal**

```
let persona = {
    nombre: "Leia",
    apellido: "Organa",
    edad: 32
}
```

**Array**

```
let nombres = [
    "Maria",
    "Sofia",
    "Sheila"
]
```

**#4** - Como cada atributo tiene un nombre, los objetos son ideales para agrupar mucha información de una sola entidad

# Objeto vs arrays

**Objeto Literal**

```
let persona = {
    nombre: "Leia",
    apellido: "Organa",
    edad: 32
}
```

**Array**

```
let nombres = [
    "Maria",
    "Sofia",
    "Sheila"
]
```

**#5 -** En cambio, los arrays son muy utilizados para armar listas de elementos similares

# Objetos literales

Los objetos literales son otro **tipo de variable** que permite guardar mucha información en una sola variable

¿Similar al array? Si...pero con sus diferencias

# Definiendo un objeto

```
let persona = {
    nombre: "Hermione",
    apellido: "Granger",
    edad: 12
}
```

# Definiendo un objeto

```
let persona = {
    nombre: "Hermione",
    apellido: "Granger",
    edad: 12,
    amigos: ["Harry", "Ron", "Ginny"]
}
```

Sí...se puede complicar....

# Uso

Cuando tengo que crear un elemento cuya información está compuesta por más de un valor y existen operaciones comunes (funciones) para todos los elementos de este tipo y sus propiedades, debe definirse como un objeto.

```
let nombre = "Harry";
let edad   = 11;
let calle  = "Privet Drive n. ° 4";
// Los variables anteriores están relacionados entre sí, entonces mejor usamos un objeto literal
const persona1 = { nombre: "Harry", edad: 39, calle: "Privet Drive n. ° 4"; }
```

```
console.log(persona1.nombre);
console.log(persona1.edad);
console.log(persona1.calle);
```

Para obtener el valor de una propiedad en un objeto utilizamos la notación punto (.): El nombre de la variable del objeto, seguido de punto y el nombre de la propiedad:

---

```
console.log(persona1["nombre"]);
console.log(persona1["edad"]);
console.log(persona1["calle"]);
```

```
Harry
11
Privet Drive n. ° 4
>
```

Otra forma de obtener el valor de una propiedad en un objeto utilizamos la notación corchetes ([]): El nombre de la variable del objeto, seguido de corchetes y dentro de ellos un string del nombre de la propiedad.

# Leyendo un objeto

```
let persona = {
     nombre: "Hermione",
     apellido: "Granger",
     edad: 12,
     amigos: ["Harry", "Ron", "Ginny"]
}
```

```
let nombrePersona = persona.nombre
let amigos = persona.amigos[0]
```

# Modificar un objeto

```
let persona = {
     nombre: "Hermione",
     apellido: "Granger",
     edad: 12,
     amigos: ["Harry", "Ron", "Ginny"]
}


persona.edad = persona.edad + 1;

persona.amigos.push("Neville");

persona.ocupacion = "Ministra de Magia";
```

# Enumerar las propiedades

A partir de ECMAScript 5, hay tres formas nativas para enumerar/recorrer las propiedades de objetos:

- `bucles for...in`
  Este método recorre todas las propiedades enumerables de un objeto y su cadena de prototipos.
  ```
  for (let prop in miObjeto[0]) {console.log(prop)};
  ```

- `Object.keys(nombreObjeto)`
  Este método devuelve un arreglo con todos los nombres de propiedades enumerables ("claves") propias (no en la cadena de prototipos) de un objeto o.
  ```
  console.log(Object.keys(miObjeto[0]));
  ```

- `Object.getOwnPropertyNames(nombreObjeto)`
  Este método devuelve un arreglo que contiene todos los nombres (enumerables o no) de las propiedades de un objeto
  ```
  console.log(Object.getOwnPropertyNames(miObjeto[0]));
  ```

# ¡A practicar!

- ★ Creá un objeto literal con nombre, apellido, edad, si tenés mascota y serie favorita.
- ★ Mostrá el objeto en consola.
- ★ Luego, agregale un atributo más llamado “comidasFavoritas” y asignale de valor 2 o más comidas que te gusten.
- ★ Hacé un nuevo **console.log()** del objeto.

# setInterval ()

El método setInterval () llama a una función o evalúa una expresión a intervalos específicos (en milisegundos).

```
setInterval(function(){ alert("Hola mundo"); }, 3000);
```

# setInterval ()

- El método setInterval () continuará llamando a la función hasta que se llame a clearInterval () o se cierre la ventana.
- El valor de ID devuelto por setInterval () se utiliza como parámetro para el método clearInterval ().
- Consejo: 1000 ms = 1 segundo.
- Para ejecutar una función solo una vez, después de un número específico de milisegundos, use el método setTimeout () .

# setTimeout ()

El método **setTimeout ()** llama a una función o evalúa una expresión después de un número específico de milisegundos.

```
setTimeout(function(){ alert("Hola Mundo"); }, 3000);
```

# setTimeout ()

- 1000 ms = 1 segundo.
- la función solo se ejecuta una vez. Si necesita repetir la ejecución, use el método setInterval () .
- el método clearTimeout () para evitar que la función se ejecute.

# ¡A practicar!

- ★ Creá una función que muestre la hora.
- ★ El reloj debe mostrarse por pantalla.
- ★ Creá el css y la estructura html necesaria.

# 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 JS podemos modificar el texto contenido de una etiqueta <h1>.

# ¿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>

# Acceder a los nodos

Existen distintos métodos para acceder a los elementos del DOM empleando en la clase Document.
Los más utilizados son:

- getElementById()
- getElementsByClassName()
- getElementsByTagName()

# Creación De Elementos

Para crear elementos se utiliza la función document.createElement(), y se debe indicar el nombre de etiqueta HTML que representará ese elemento.
Luego debe agregarse como hijo el nodo creado con appendChild(), al body u a otro nodo del documento actual.

# Creación De Elementos



```
// Crear nodo de tipo  Elemento, etiqueta p
let parrafo = document.createElement("p");
//  Insertar  HTML interno
parrafo.innerHTML = "<h2>¡Hola  Mundo!</h2>";
// Añadir el nodo  Element como hijo de body
document.body.appendChild(parrafo);
```

# Eliminar elementos

Se pueden eliminar nodos existentes y nuevos.

El método **removeChild()** permite eliminar nodos hijos a cualquier nodo con tan sólo pasarle las referencias del nodo hijo [a] eliminar y su correspondiente padre.

# Eliminar elementos

```
let parrafo = document.getElementById("parrafo1");
//Elminando el propio elemento, referenciando al padre
parrafo.parentNode.removeChild(parrafo);

let paises = document.getElementsByClassName("paises");
//Eliminando el primer elemento de clase paises
paises[0].parentNode.removeChild(paises[0])
```

# Obtener datos de inputs

Para obtener o modificar datos de un formulario HTML desde JS, podemos hacerlo mediante el DOM. Accediendo a la propiedad value de cada input identificado con un ID.

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

```
//CODIGO JS
document.getElementById("nombre").value = "Harry";
document.getElementById("edad").value  = 13;
```

# Delegación de eventos

Delegación de eventos es un mecanismo a través del cual evitamos asignar *event listeners* a múltiples nodos específicos del DOM, asignando un *event listener* a solo un nodo padre que contiene el resto de estos nodos.

La ventaja de usar delegación de eventos es que en el caso hipotético de tener ***n*** elementos con la clase `btn`, solo hemos registrado un *event listener* para todos estos elementos, mientras que sin delegación de eventos debemos registrar ***n*** *event listeners*, es decir uno por cada nodo.

# Delegación de eventos

La delegación de eventos es un patrón útil porque puede escuchar eventos en múltiples elementos usando un controlador de eventos.

Hacer que la delegación de eventos funcione requiere 3 pasos:

1. Determinar el padre de los elementos para vigilar los eventos.
2. Adjunte el detector de eventos al elemento principal.
3. Use **event.target** para seleccionar los elementos de destino.

# Glosario

**event.preventDefault:** Cancela el comportamiento por defecto del evento si este es cancelable, sin detener el resto del funcionamiento del evento, es decir, puede ser llamado de nuevo.

**parentElement**: La propiedad **parentElement** devuelve el elemento padre del elemento especificado.

# Glosario

**Element.classList:** La propiedad de sólo lectura **Element.classList** devuelve una colección activa de DOMTokenList de los atributos de clase del elemento.
Usar classList es una forma práctica de acceder a la lista de clases de un elemento como una cadena de texto delimitada por espacios a través de element.className.

**contains**(String): Comprueba si la clase indicada existe en el atributo de clase del elemento.

UTN.BA

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES