BOOTCAMP

Generación T

streambe

Repaso General de Fundamentos de JavaScript + DOM

streambe GENERACIÓN T

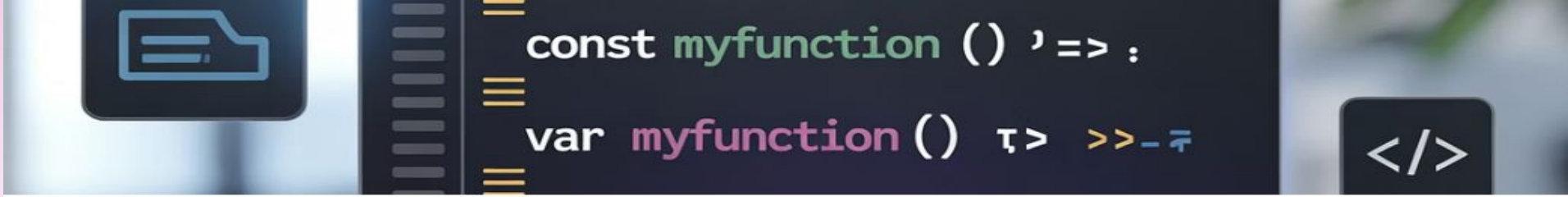
¿Qué es una función clásica?

Las funciones clásicas en JavaScript son bloques de código reutilizables diseñados para realizar tareas específicas y devolver resultados.

Características principales:

- Permiten encapsular lógica compleja en unidades manejables
- Mejoran la modularidad y mantenibilidad del código
- Tienen su propio contexto de ejecución (this)


```
function calcularArea(base, altura) {
  return base * altura / 2;
}

const resultado = calcularArea(5, 3);
```



Sintaxis y ejemplo de función clásica

</>

Declaración de función

```
function saludar(nombre) {
  return "Hola, " + nombre;
}
```

Se eleva (hoisting) en el ámbito

ƒ(x)

Expresión de función

```
const saludar =
function(nombre) {
  return "Hola, " + nombre;
};
```

No se eleva completamente

Método de objeto

```
const persona = {
  saludar: function(nombre)
{
    return "Hola, " +
nombre;
  }
};
```

¿Qué es una arrow function?

Las arrow functions (o funciones flecha) representan una sintaxis más moderna y concisa para escribir funciones en JavaScript.

Introducidas con ES6 en 2015, estas funciones se caracterizan por:

- Sintaxis más breve que reduce el código boilerplate
- Comportamiento lexical de **this** (heredado del contexto)
- Ausencia de objeto **arguments** propio

Las arrow functions simplificaron significativamente la escritura de callbacks y funciones anónimas.

Sintaxis y ejemplo de arrow function

1 Sintaxis básica

```
// Con parámetros y bloque
const suma = (a, b) => {
  return a + b;
};

// Retorno implícito (sin llaves)
const suma = (a, b) => a + b;

// Un solo parámetro (sin paréntesis)
const cuadrado = x => x * x;
```

2 Casos especiales

```
// Sin parámetros
const saludar = () => "Hola mundo";

// Retornando un objeto literal
const crearPersona = (nombre, edad) => ({
  nombre,
  edad
});
```

Arrays vs Objetos: Estructuras de datos fundamentales

Arrays

Colecciones **ordenadas** de elementos accesibles por índice numérico.

```
const frutas = ['manzana', 'naranja',
'plátano'];
frutas[1]; // 'naranja'
```

Objetos

Colecciones de pares **clave-valor** sin orden específico.

```
const persona = {nombre: 'Ana', edad: 28};
persona.nombre; // 'Ana'
```

Métodos comunes en Arrays

- map(), filter(), reduce()
- push(), pop(), shift(), unshift()
- forEach(), find(), some(), every()

Métodos comunes en Objetos

- Object.keys(), Object.values()
- Object.entries(), Object.assign()
- Object.freeze(), Object.seal()

Recorriendo Arrays y Objetos con Bucles

Bucles para Arrays

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

// for...of (valores)
for (const elemento of array) {
  console.log(elemento);
}
```

Bucles para Objetos

```
// for...in (claves)
for (const clave in objeto) {
  console.log(clave, objeto[clave]);
}

// Object.entries()
Object.entries(objeto).forEach(
  ([clave, valor]) => console.log(clave,
valor)
);
```

Métodos funcionales (Arrays)

- forEach(): Ejecuta función por cada elemento
- Más declarativo y legible que bucles tradicionales
- Menos propenso a errores de índices

Consideraciones de rendimiento

- for clásico: Mayor rendimiento en arrays grandes
- for...of: Mejor legibilidad con buen rendimiento
- for...in: Evitar en arrays (incluye propiedades heredadas)

Manipulación del DOM: Selección de Elementos

1 Por ID

Selecciona un elemento único mediante su identificador.

```
const elemento =
document.getElementById(
'miID');
elemento.style.color =
'red';
```

2 Por Clase

Obtiene una colección de elementos que comparten una clase.

```
const elementos =
document.getElementsByCl
assName('miClase');
// O con mayor
flexibilidad
const elementos =
document.querySelectorAl
l('.miClase');
```

3 Por Etiqueta

Selecciona todos los elementos de un tipo específico.

```
const parrafos =
document.getElementsByTa
gName('p');
// Alternativa con
querySelector
const primerParrafo =
document.querySelector('
p');
```

Los selectores CSS avanzados con **querySelector()** y **querySelectorAll()** ofrecen mayor flexibilidad para selecciones complejas como *'div.container > p:first-child'*.

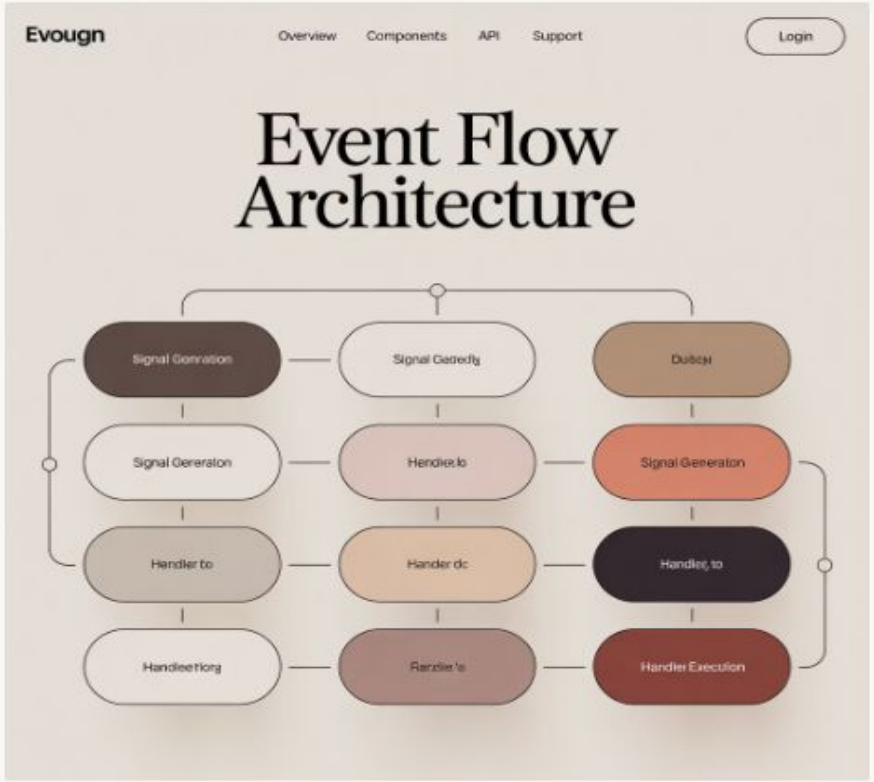
Eventos!

¿Cómo Funcionan los Eventos?

El navegador actúa como un vigilante constante, monitoreando todas las interacciones posibles:

- Genera señales cuando detecta una acción
- Asocia cada acción con un manejador específico
- Ejecuta el código correspondiente automáticamente

Este sistema permite crear interfaces reactivas sin necesidad de recargar la página.



Tipos Comunes de Eventos

Eventos de Ratón

click, dblclick, mouseover, mouseout y mousedown capturan diferentes interacciones del puntero con elementos de la página.

Eventos de Teclado

keydown, keyup y keypress permiten responder a las pulsaciones de teclas por parte del usuario.

Eventos de Formulario

submit, change y focus controlan las interacciones con formularios y campos de entrada.

Eventos de Ventana

load, resize y scroll detectan cambios en la ventana del navegador o la carga de la página.

Tipos De Eventos

Eventos Más Comunes

Evento	Descripción
onchange	Modificación de un elemento HTML
onclick	Usuario hace *click* sobre un elemento HTML
mouseover	Usuario mueve el *mouse* sobre un elemento HTML
mouseout	Usuario quita el *mouse* del elemento HTML
keydown	Usuario presiona una tecla
load	*Browser* termina de cargar la página

Detectores y Manejadores de Eventos

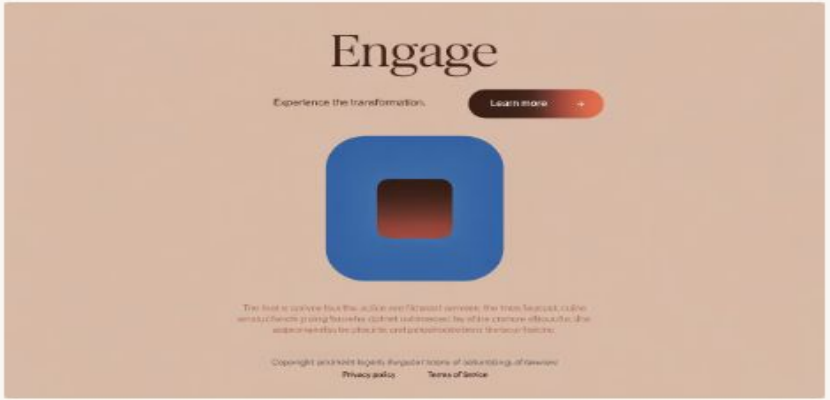
Conceptos Clave

- **Detector:** Observa si ocurre un evento específico
- **Manejador:** Función que se ejecuta cuando se detecta el evento
- **addEventListener:** Método para conectar eventos con funciones

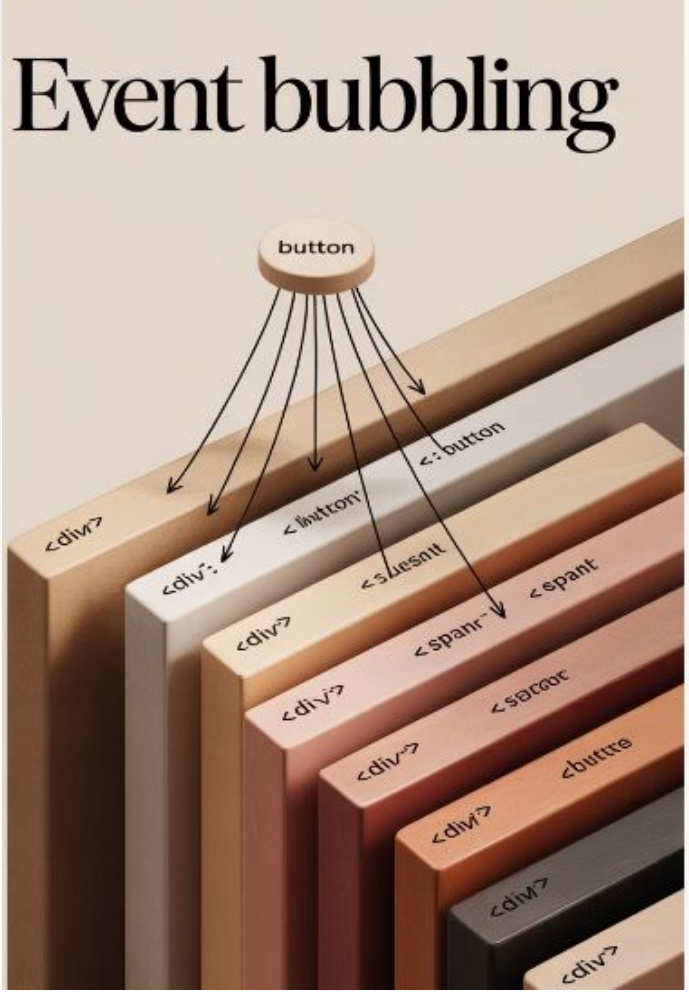

```
// Sintaxis básica
elemento.addEventListener('evento', función);

// Ejemplo práctico
button.addEventListener('click', function() {
  console.log('¡Botón pulsado!');
});
```

Ejemplo Práctico: Click en un Botón

```
// HTML
<button id="miBoton">Cámbiar Color</button>
<div id="cuadrado"></div>

// CSS
#cuadrado {
  width: 100px;
  height: 100px;
  background-color: blue;
}

// JavaScript
document.getElementById('miBoton')
  .addEventListener('click', function() {
    const cuadrado =
      document.getElementById('cuadrado');
    cuadrado.style.backgroundColor = 'red';
  });
```



Este ejemplo muestra cómo un evento simple puede crear interactividad instantánea, cambiando propiedades visuales sin recargar la página.



Burbujeo de Eventos (Event Bubbling)

Propagación Ascendente

Un evento se dispara primero en el elemento más específico (p. ej., un botón)

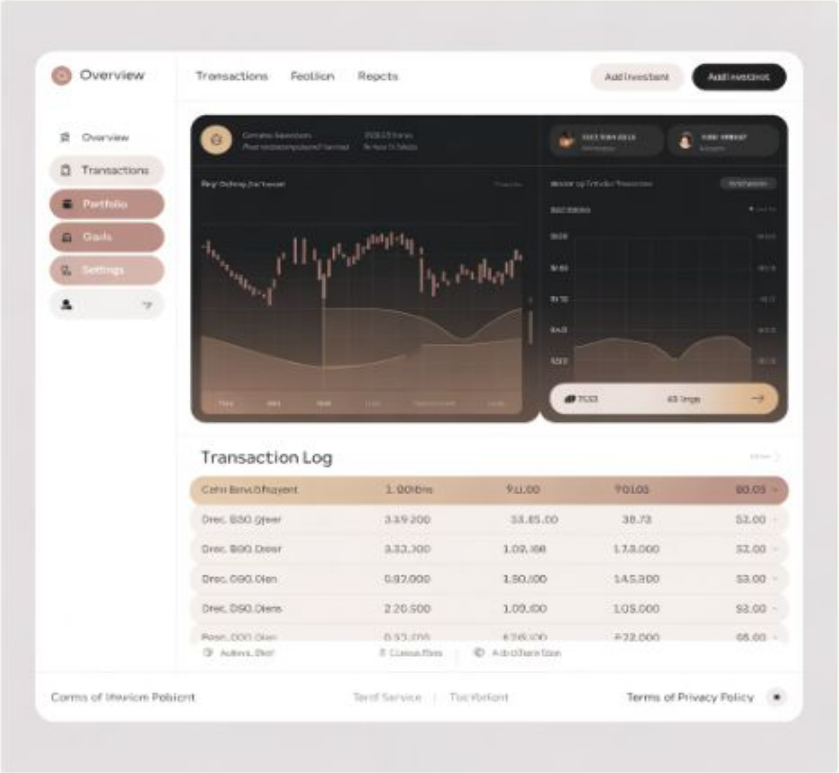
Ascenso por el DOM

Luego se propaga hacia arriba, activando manejadores en elementos padres

Control con stopPropagation()

Podemos detener este burbujeo usando event.stopPropagation() si es necesario

¿Por Qué Son Importantes los Eventos?



Experiencias Interactivas

Transforman sitios estáticos en aplicaciones dinámicas que responden instantáneamente

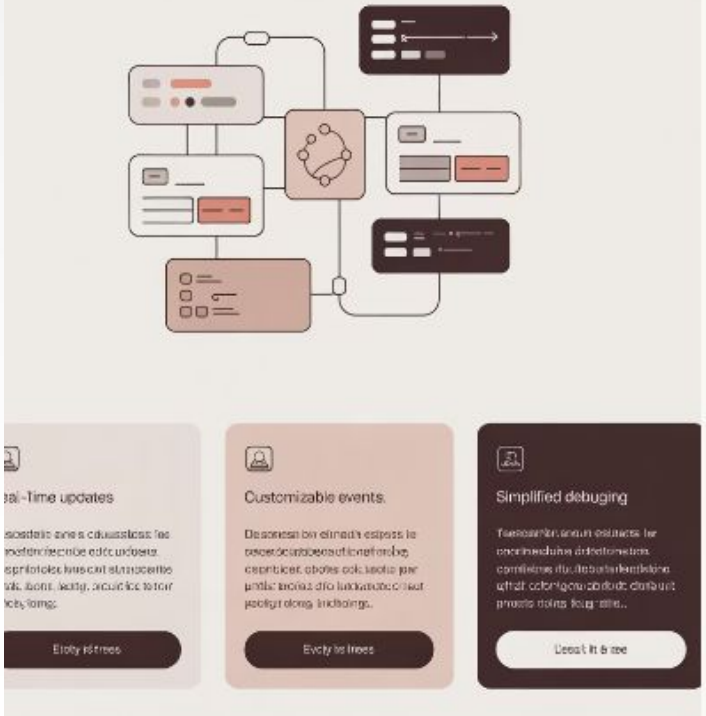
Interfaz Receptiva

Permiten feedback inmediato sin recargas de página completa

Arquitectura Moderna

Son fundamentales en frameworks como React, Vue y Angular

Orchestrate your interactions



Resumen

Señales Digitales

Los eventos son notificaciones que indican que algo relevante ha ocurrido en el navegador

Código Reactivo

Permiten ejecutar funciones específicas en respuesta a acciones del usuario o del sistema

Web Interactiva

Son esenciales para crear experiencias web modernas, fluidas y centradas en el usuario

¡Dominar los eventos es crucial para cualquier desarrollador frontend!

Muchas gracias.

streambe GENERACIÓN T

Nuestras Redes

www.generaciont.org
www.streambe.com
www.instagram.com/generaciont_ar
www.tiktok.com/@generaciont
generaciont@generaciont.org
Cel: 11 61331747

GENERACIÓN T

streambe