Manual Imprescindible

CURSO DE JAVASCRIPT

Astor de Caso Parra

ANAYA MULTIMEDIA

Índice de contenidos

2 Variables y tipos de datos

En este capítulo aprenderás:

- Declarar variables y constantes.
- Los tipos de datos disponibles en JavaScript.
- Los valores especiales de las variables.

Variables

Una variable es un espacio de memoria donde almacenamos temporalmente un dato que utilizamos en las operaciones dentro de nuestro código, como, por ejemplo, el resultado de una suma. Ese dato podrá ser recuperado en cualquier momento e incluso ser modificado.

En el código, cada variable estará identificada por un nombre a nuestra elección y podrá estar formado por caracteres alfanuméricos, el carácter guión bajo o subrayado (_) y el símbolo de dólar ($). La única restricción es que no puede comenzar por un número. También es importante recordar que se distingue entre mayúsculas y minúsculas, por lo que la variable `suma` no será la misma que la variable `SUMA`. Puede ver ejemplos en la tabla 2.1.

Tabla 2.1. Ejemplos de nombres de variables.

Nombre válido	Nombre no válido
dos_numeros	2_numeros
nombre1	nombre-1
_Documento	@Documento
nombreApellidos	nombre&apellidos
$dinero	€dinero

También debe tener en cuenta que el nombre de una variable no puede coincidir con el de una palabra reservada de JavaScript (véase la tabla 2.2) ni con el de otro elemento que hayamos definido (funciones u objetos).

Tabla 2.2. Lista de palabras reservadas ECMAScript.

Palabras reservadas			
abstract	else	int	synchronized
await	enum	interface	this
boolean	export	let	throw
break	extends	long	throws
byte	false	native	transient
case	final	new	true
catch	finally	null	try
char	float	package	typeof
class	for	private	var
const	function	protected	void

Valor operando	Ejemplo	Resultado
`var x = "hola";`	`var y = !x;`	y vale `false`
`var x = "false";`	`var y = !x;`	y vale `false`
`var x = undefined;`	`var y = !x;`	y vale `true`

Operador Y (AND)

Con este operador vamos a poder utilizar variables de cualquier tipo, pero no vamos a obtener siempre un resultado booleano. En estos casos, el operador va a ir examinando cada operando mientras se evalúen como `true` y entonces devuelve el último como resultado, sea del tipo que sea. Si alguno se traduce como `false`, entonces lo devuelve y deja de mirar el resto.

Tabla 3.26. Operador lógico Y con variables no booleanas.

Valor operando 1	Valor operando 2	Ejemplo	Resultado
`var x = true;`	`var y = 0;`	`var z = x && y;`	z vale `0` (primer valor `false`)
`var x = 0;`	`var y = true;`	`var z = x && y;`	z vale `0` (primer valor `false`)
`var x = true;`	`var y = 1;`	`var z = x && y;`	z vale `1` (último valor `true`)
`var x = 1;`	`var y = true;`	`var z = x && y;`	z vale `true` (último valor `true`)
`var x = false;`	`var y = 0;`	`var z = x && y;`	z vale `false` (primer valor `false`)
`var x = 0;`	`var y = 1;`	`var z = x && y;`	z vale `0` (primer valor `false`)
`var x = true;`	`var y = "";`	`var z = x && y;`	z vale `""` (primer valor `false`)
`var x = "";`	`var y = true;`	`var z = x && y;`	z vale `""` (primer valor `false`)
`var x = true;`	`var y = "hola";`	`var z = x && y;`	z vale `"hola"` (último valor `true`)
`var x = "hola";`	`var y = true;`	`var z = x && y;`	z vale `true` (último valor `true`)
`var x = true;`	`vary = undefined;`	`var z = x && y;`	z vale `undefined` (primer valor `false`)
`var x = null;`	`var y = true;`	`var z = x && y;`	z vale `null` (primer valor `false`)
`var x = "";`	`var y = "hola";`	`var z = x && y;`	z vale `""` (primer valor `false`)
`var x = NaN;`	`var y ="hola";`	`var z = x && y;`	z vale `NaN` (primer valor `false`)
`var x = "hola";`	`var y = "adiós";`	`var z = x && y;`	z vale `"adiós"` (último valor `true`)

Operador O (OR)

El comportamiento es similar al operador AND, salvo que devuelve como resultado el primer operando interpretado como `true` o el último en otro caso.

Tabla 3.27. Operador lógico O con variables no booleanas.

Valor operando 1	Valor operando 2	Ejemplo	Resultado
`var x = true;`	`var y = 0;`	`var z = x || y;`	z vale `true` (primer valor `true`)
`var x = 0;`	`var y = true;`	`var z = x || y;`	z vale `true` (primer valor `true`)
`var x = true;`	`var y = 1;`	`var z = x || y;`	z vale `true` (primer valor `true`)
`var x = 1;`	`var y = true;`	`var z = x || y;`	z vale `1` (primer valor `true`)
`var x = false;`	`var y = 0;`	`var z = x || y;`	z vale `0` (último valor `false`)
`var x = 0;`	`var y = false;`	`var z = x || y;`	z vale `false` (último valor `false`)
`var x = false;`	`var y = 1;`	`var z = x || y;`	z vale `1` (primer valor `true`)
`var x = 1;`	`var y = false;`	`var z = x || y;`	z vale `1` (primer valor `true`)
`var x = 1;`	`var y = 0;`	`var z = x || y;`	z vale `1` (primer valor `true`)
`var x = "";`	`var y = true;`	`var z = x || y;`	z vale `true` (primer valor `true`)
`var x = "";`	`var y = 0;`	`var z = x || y;`	z vale `0` (último valor `false`)
`var x = "hola";`	`var y = true;`	`var z = x || y;`	z vale `"hola"` (primer valor `true`)
`var x = true;`	`var y = undefined;`	`var z = x || y;`	z vale `true` (primer valor `true`)
`var x = null;`	`var y = true;`	`var z = x || y;`	z vale `true` (primer valor `true`)
`var x = "";`	`var y = "hola";`	`var z = x || y;`	z vale `"hola"` (primer valor `true`)
`var x = NaN;`	`var y = "hola";`	`var z = x || y;`	z vale `"hola"` (primer valor `true`)
`var x = "hola";`	`var y = "adiós";`	`var z = x || y;`	z vale `"hola"` (primer valor `true`)

```
    set marca(valorMarca) {
       this._marca = valorMarca;
    }
};
// Utilizamos el setter
unCoche.marca = "Fiat";
// Accedemos al getter
console.log(unCoche.marca); // Mostraría "Fiat"
```

ADVERTENCIA:

El nombre del método `set` *no puede coincidir con el de la propiedad a la que se vincula, pues crearíamos un bucle infinito de llamadas al método (recuerde, una asignación invoca al setter).*

Como ha visto en los ejemplos, hemos definido una pseudopropiedad `marca` que realmente no está definida en el objeto (`this.marca`), pero hemos sido capaces de utilizarla como si fuera una propiedad normal. El *getter* y *setter* quedan vinculados en este caso con una propiedad real `_marca`, a la cual podríamos acceder con total normalidad (`unCoche._marca`), pero la finalidad de estos métodos es "esconder" en cierta medida el nombre real. Además, también nos pueden ser útiles si la propiedad necesita algún tipo de cálculo o tratamiento en alguna de las dos operaciones.

```
// Objeto con getter y setter
var unCoche = {
   get marca() {
      return `Este coche es: ${this._marca}`;
   },
   set marca(valorMarca) {
      this._marca = `--- ${valorMarca} ---`;
   }
};
// Utilizamos el setter
unCoche.marca = "Fiat";
// Accedemos al getter
console.log(unCoche.marca); // Mostraría "Este coche es: --- Fiat ---"
```

Para terminar, quería mencionarle que también es posible eliminar un *getter* y *setter* mediante el operador `delete` como si fuera una propiedad más.

```
// Accedemos al getter
console.log(unCoche.marca); // Mostraría "Este coche es: --- Fiat ---"
// Eliminamos la pseudopropiedad
delete unCoche.marca;
console.log(unCoche.marca); // Mostraría undefined
```

Puede que esté pensando: "¿Y qué ocurre si vuelvo a asignar un valor a la propiedad `marca`?". Pues vamos a verlo.


```
// Asignamos valor a la propiedad
unCoche.marca = "Fiat";
// Accedemos a la propiedad
console.log(unCoche.marca); // Mostraría "Fiat"
```

Pero, si lo hemos eliminado, ¿cómo es que no da error? Si observa atento el valor recuperado, podrá ver que ya no es el mismo texto que habíamos definido con el *getter* y *setter*, sino que es simplemente el valor sin alterar. Esto nos da la pista para concluir que lo que hemos hecho en realidad es añadir de nuevo `marca` como una propiedad normal, sin métodos `get` y `set`.

Otros operadores sobre un objeto

Aunque en las secciones anteriores han ido apareciendo la mayoría de los operadores que nos ofrece JavaScript para trabajar con los objetos (`this`, `new`, etc.), aún quedan un par más para ayudarle a tener más control sobre los objetos.

- `instanceof`: Nos dice si la instancia especificada proviene del objeto que indiquemos. Esto es, nos indica mediante un valor `true` o `false` si la instancia ha sido creada a partir de un objeto concreto. Veamos su sintaxis y funcionamiento con un pequeño ejemplo:

```
// Objeto definido con llaves
var coche1 = {
  marca: "Fiat"
};
// Objeto definido con function
function MiCoche(valorMarca) {
  // Definición de propiedad
  this.marca = valorMarca;
}
// Definimos instancias
var coche2 = new MiCoche("Fiat");
console.log(coche1 instanceof MiCoche); // false, objeto definido con llaves
console.log(coche2 instanceof MiCoche); // true, es instancia de MiCoche
```

- `in`: Nos permite saber si una propiedad o método existe en el objeto o instancia que indiquemos, devolviendo un valor `true` o `false`. El nombre a comprobar debe estar escrito como un *string*. Si lo usamos contra un objeto, solamente funcionará con los que vienen predefinidos en JavaScript (detallados en los siguientes capítulos).

```
// Objeto definido con llaves
var coche1 = {
  marca: "Fiat"
};
// Objeto definido con function
```

7 Objetos básicos de JavaScript

En este capítulo aprenderás:

- Manejar los objetos fundamentales de JavaScript.
- Diferenciar los métodos de objeto y de instancia.
- El objeto JSON.

Dentro del núcleo de JavaScript, existen varios objetos que ya están definidos y tienen una serie de funcionalidades tremendamente útiles en muchos casos. Para no desarrollar un apartado demasiado extenso, he preferido dividirlos en varios grupos y dedicarle un capítulo a cada uno de esos grupos:

1. **Objetos básicos:** Les he dado este nombre ya que son objetos sin una excesiva complejidad o son un envoltorio (*wrapper*) de los tipos de datos predefinidos (booleano, número y cadena o *string*), es decir, crear una instancia de uno de estos objetos mediante la sentencia `new` equivale a declarar una variable directamente con el valor. Son los que veremos en este capítulo.
2. **Objetos intermedios:** Aquí se engloban objetos que tienen cierta personalidad propia y utilizan tipos de datos que aún no hemos visto.
3. **Objetos avanzados:** No se asuste por el nombre. La única diferencia respecto a los grupos anteriores es que manejan estructuras de datos que aún no hemos presentado y por eso requieren una explicación algo más extensa.

De cada uno de los objetos expondremos el constructor, propiedades y métodos que tengan asociados (al menos, los principales), así como ejemplos de algunos de ellos para mejorar su comprensión o utilidad.

NOTA:

Algunos métodos han sido introducidos en versiones posteriores a ES6, por lo que aparecerán marcados entre corchetes (por ejemplo, [ES7]) para que tenga en cuenta que, si los utiliza, el entorno donde lo utilice debe soportar la especificación correspondiente de ECMAScript.

Le adelanto que todos los objetos predefinidos en JavaScript disponen el método `toString()`, que devuelve el objeto expresado como una cadena, realizando para ello conversiones implícitas si es necesario. Para hacer uso de él, basta con hacer la llamada desde la instancia que nos interese: `miInstancia.toString()`.

Objeto Boolean

Como puede imaginar, este objeto expresa un valor como un booleano, o lo que es lo mismo, como `true` o `false`.

Objeto Array

¡Vamos por fin con este objeto que tantas veces hemos nombrado páginas atrás!

Las variables de JavaScript normalmente almacenan un único valor, el cual podemos recuperar cuando utilizamos su nombre dentro de una expresión. Esto se nos puede quedar corto en determinadas ocasiones como si, por ejemplo, quisiéramos almacenar por separado los platos del menú de un restaurante para después mostrarlos en nuestra página. Gracias a los *arrays* podremos almacenar todos estos platos en una única variable como si esta tuviera huecos o casillas que se rellenan con los valores, permitiéndonos acceder a cada plato individual con tan solo indicar cuál es su hueco, es decir, la posición dentro del *array*. A esta posición se le denomina índice.

Constructor

Para crear un *array* tenemos varios constructores:

1. `new Array()`: Crea un *array* vacío, sin huecos donde almacenar nuestros datos. Esto es útil cuando no sabemos la longitud inicial que tendrá el *array* cuando lo utilicemos en nuestro código.

```
// Objeto Array
var miArray = new Array(); // Vacío y sin posiciones
```

2. `new Array(longitud)`: En este caso el *array* se crea con tantos huecos como indique el parámetro `longitud`. Cada hueco tendrá un valor `undefined`.

```
// Objeto Array
var miArray = new Array(5); // 5 posiciones vacías
```

3. `new Array(valor1, valor2, ..., valorN)`: Con este constructor crearemos un *array* con tantas posiciones como valores hayamos pasado como parámetros y, además, dichos huecos irán rellenos con el valor correspondiente. Tenga cuidado, pues, si únicamente indica un valor y es de tipo numérico, JavaScript lo interpretará como el constructor anterior.

```
// Objeto Array
var array1 = new Array("5");           // 1 posición, 1 valor string
var array2 = new Array("a", "b", "c"); // 3 posiciones, valores string
var array3 = new Array(1, 2, 3);       // 3 posiciones, valores number
var array4 = new Array("a", 2, false); // 3 posiciones, valores de tipos
                                          distintos
```


4. Mediante el uso de corchetes (`[]` o `[valor1, valor2, ..., valorN]`): No es propiamente un constructor, sino que equivaldría a crear directamente una instancia, como ocurría con los objetos definidos con llaves. Obtendremos un *array* vacío si no se indican valores o bien uno relleno con los valores pasados como parámetros.

```
// Creación de instancias
var array1 = [];              // Vacío y sin posiciones
var array2 = [5];             // 1 posición, valor number
var array3 = ["a", "b", "c"]; // 3 posiciones, valores string
var array4 = [1, 2, 3];       // 3 posiciones, valores number
var array5 = ["a", 2, false]; // 3 posiciones, valores de tipos distintos
```

El tipo de datos de los valores almacenados en un *array* puede ser cualquiera, incluso puede ser distinto en cada una de las posiciones. En otros lenguajes, los *arrays* están forzados a que los datos sean siempre del mismo tipo (entero, booleano, etc.).

TRUCO:

Podemos crear una instancia (mediante corchetes) con posiciones vacías simplemente poniendo tantas comas como huecos necesitemos, omitiendo los valores. Equivaldría al constructor `new Array(longitud)`.

```
// Creación de instancia con posiciones vacías
var array1 = [ , ]; // 1 posición
var array2 = [ , , , ]; // 3 posiciones
// Mostramos valores
console.log(array1);    // [undefined]
console.log(array2);    // [undefined, undefined, undefined]
```

Propiedades

El objeto `Array` solo dispone de una propiedad:

- `length`: Nos indicará la longitud o número de elementos del *array*. Aquí hay que prestar atención, puesto que no influye el que los huecos tengan valor o no, sino que basta con que la posición exista y sea accesible.

```
// Creación de instancias
var array1 = new Array();
var array2 = new Array(3);
var array3 = ["a", "b"];
// Mostramos valores
console.log(array1.length); // 0
console.log(array2.length); // 3
console.log(array3.length); // 2
```

Para poder experimentar la asincronía y no tener solamente pseudocódigo, le presento una función de JavaScript que nos permitirá simular la respuesta a una llamada asíncrona.

- `setTimeout(fnCallback, milisegundos, param1, param2, ...paramN)`: Ejecuta una función *callback* pasado el intervalo de tiempo indicado en milisegundos. Si este no se especifica, tomará valor cero por defecto. Los parámetros siguientes son opcionales y son pasados a la función *callback* como argumentos.

```
// Declaración de función
function mostrarMensaje() {
   console.log("Hola");
}
// Mostraría "Hola" pasados 5 segundos
setTimeout(mostrarMensaje, 5000);
// Mostraría "Hola otra vez" pasados 10 segundos
setTimeout(() => console.log("Hola otra vez"), 10000);
// Mostraría "Hola de nuevo" pasados 3 segundos
setTimeout(console.log, 3000, "Hola de nuevo");
```

Puede comprobar que es una función sencilla de entender y de manejar, así que vamos a volver a modificar nuestro ejemplo para que se realice el sumatorio con cinco segundos de retardo, simulando la espera de respuesta de la base de datos.

```
// Suma de números, modo asíncrono
// Declaración de función
function sumar(acumulado, numero) {
   return acumulado + numero;
}
// Función de callback
function calcularSuma(array) {
   var resultado = array.reduce(sumar, 0);
   console.log(resultado); // 6
}
// Código asíncrono
function obtenerNumeros(fnCallback) {
   numeros = [1, 2, 3];
   setTimeout(() => fnCallback(numeros), 5000);
}
// Obtención de datos y callback cuando termine
obtenerNumeros(calcularSuma);
console.log("Esperando el resultado...");
// Se mostrarán los mensajes:
// "Esperando el resultado..."
// 6 (5 segundos más tarde)
```

¡El experimento es todo un éxito! Con esto hemos podido comprobar que el flujo de ejecución no se detiene en la llamada a `obtenerNumeros`, puesto que la instrucción siguiente se ejecuta al instante (el mensaje de espera) y el *callback* `calcularSuma` lo hace segundos más tarde.

Ahora está preparado para seguir profundizando en la programación asíncrona con JavaScript.

ADVERTENCIA:

Tenga en cuenta que en las operaciones asíncronas el orden de las respuestas no siempre coincide con el que han sido escritas las instrucciones.

```
// Declaración de función
function mostrarMensaje(texto) {
   console.log(texto);
}
// Varias operaciones asíncronas
setTimeout(mostrarMensaje, 5000, "Buenos días");
setTimeout(mostrarMensaje, 10000, "Buenas tardes");
setTimeout(mostrarMensaje, 3000, "Buenas noches");
// Se mostrarán los mensajes en este orden:
// "Buenas noches", a los 3 segundos
// "Buenos días", 2 segundos después (5 en total)
// "Buenas tardes", a los 5 segundos (10 en total)
```

Promesas

Las promesas surgen como la interpretación de JavaScript para manejar peticiones asíncronas y que se integraron de forma nativa en ECMAScript 6, por lo que no hay que recurrir a librerías adicionales. Además, incluyen mecanismos para conocer y controlar el resultado de la misma, que puede cumplirse o no, como en la vida real. 😐 Cuando una promesa se cumple, decimos que se resuelve (*resolve*) y, en caso contrario, que se rechaza (*reject*).

Estados de una promesa

Cuando se trabaja con este concepto de asincronía, podemos pasar por varias fases durante su ciclo de vida.

- **Pendiente (*pending*):** Corresponde a su estado inicial, cuando todavía no conocemos su resultado.
- **Resuelta (*fulfilled*):** Indica que la promesa se ha cumplido y obtendremos un valor a cambio que será el resultado de la ejecución.
- **Rechazada (*rejected*):** Nos hace saber que la ejecución ha fallado y nos proporcionará el motivo del error.

14 Eventos

En este capítulo aprenderás:

- Reaccionar a eventos desde el *front-end*.
- El objeto `Event`.
- Controlar la propagación de eventos.
- Trabajar con eventos desde el *back-end*.

Cuando alguien visita nuestra página o sitio web, no se limita simplemente a leer el texto que contiene o a contemplar las imágenes que hayamos insertado, sino que llega un punto en que el usuario necesita interactuar con nuestra página, como por ejemplo darle al botón de compartir en redes sociales, y nosotros tenemos que ser capaces de preparar nuestra página para reconocer esas interacciones y realizar las operaciones que sean necesarias. A estas acciones que puede hacer el usuario se les denomina eventos y engloban cosas como pasar simplemente el puntero del ratón por encima de una imagen o hacer clic sobre un botón.

JavaScript es capaz de detectar estos eventos y también nos permite asociarles unas instrucciones que se ejecutarán cuando se produzcan.

Este escenario que hemos explicado sucede únicamente en un entorno *front-end*, pero los eventos son algo que también están presentes en *back-end*, aunque ahí se desencadenan desde el propio código. En este capítulo veremos cómo tratar los eventos en ambos entornos.

NOTA:

Los eventos se consideran operaciones asíncronas, ya que no sabemos cuándo van a suceder.

Eventos en *front-end*

En JavaScript existe una amplia variedad de eventos que podremos utilizar a nuestro antojo para crear las situaciones que queramos o conseguir un efecto concreto. La tabla 14.1 le mostrará los eventos principales y la acción que los genera.

Tabla 14.1. Lista de eventos en *front-end*.

Evento	Origen
`Click`	Se hace clic sobre un elemento.
`DblClick`	Se hace doble clic sobre un elemento.
`MouseDown`	Se hace clic con un botón cualquiera del ratón.
`MouseUp`	Se libera un botón del ratón que estaba pulsado.
`MouseMove`	Se mueve el puntero por la pantalla.
`MouseOver`	El puntero entra en el área que ocupa un elemento (se pasa por encima).
`MouseOut`	El puntero sale del área que ocupa un elemento (se quita de encima).
`KeyPress`	Una tecla es pulsada. Se genera después de `KeyDown`.
`KeyDown`	Una tecla es pulsada sin soltarla.

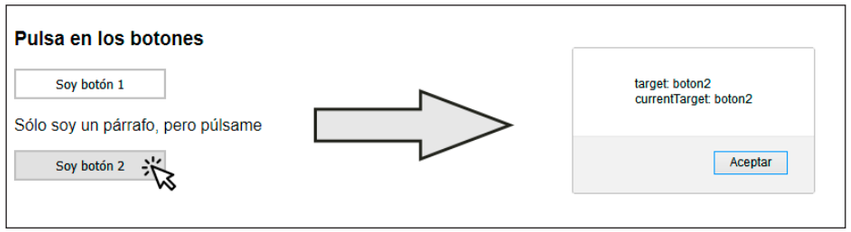

Figura 14.5. Información al hacer clic sobre **boton2**.

Tabla 14.3. Propiedades de un evento de ratón.

Propiedad	Tipo	Descripción
`altKey`	Booleano	Devuelve `true` cuando se ha presionado la tecla **Alt**.
`button`	Entero	Indica el botón del ratón que ha sido pulsado. (Ver tabla 14.4).
`clientX`	Entero	Indica la posición horizontal del cursor donde se ha hecho clic, dentro del área disponible en el documento.
`clienteY`	Entero	Indica la posición vertical del cursor donde se ha hecho clic, dentro del área disponible en el documento.
`ctrlKey`	Booleano	Devuelve `true` cuando se ha presionado la tecla **Control**.
`type`	*String*	Indica el tipo de evento que se ha producido (`click`, `mouseover`, etc.).
`screenX`	Entero	Indica la posición horizontal del cursor donde se ha hecho clic, dentro de la resolución de pantalla del usuario.
`screenY`	Entero	Indica la posición vertical del cursor donde se ha hecho clic, dentro de la resolución de pantalla del usuario.
`shiftKey`	Booleano	Devuelve `true` cuando se ha presionado la tecla **Mayús**.

Tabla 14.4. Valores de la propiedad `button`.

Valor	Descripción
`0`	Botón izquierdo del ratón.
`1`	Botón central del ratón.
`2`	Botón derecho del ratón.
`3`	Para un cuarto botón.
`4`	Para un quinto botón.

NOTA:

Si tenemos el ratón configurado para zurdos, los valores de la tabla 14.4 pueden mostrarse en otro orden.

Métodos

Ahora que conoce un poco más a los traviesos eventos, le presento algunos métodos de `Event` que nos proporcionan un mayor control sobre ellos.

- `preventDefault()`: Cancela un evento, evitando que se ejecute su acción por defecto. Esto no evita que el evento se siga propagando. Veamos un ejemplo con un `checkbox`, cuya acción por defecto en un evento `click` es marcarse o desmarcarse.

```
<HTML>
<BODY>
  <DIV ID="capa1">
     <FORM ID="formulario1">
     <INPUT
        TYPE="checkbox"
        ID="check1"
     /> Confirmar
     </FORM>
  </DIV>
  <SCRIPT TYPE="text/javascript">
     // Declaración de función
     function cancelarEvento(evento) {
        evento.preventDefault();
        console.log(`Evento ${evento.type} cancelado`);
     }
     // Asignación de manejadores
     document
        .getElementById("check1")
        .onclick = cancelarEvento;
     document
        .getElementById("capa1")
        .onclick = () => console.log("Clic en capa1");
  </SCRIPT>
</BODY>
</HTML>
```

Si hacemos clic sobre **check1**, veremos que no se selecciona nunca, ya que hemos cancelado su acción por defecto, pero sí se muestran por consola tanto el mensaje de cancelación como el mensaje de la **capa1** porque se propaga el evento hasta ella.

- `stopPropagation()`: Detiene la propagación de un evento hacia niveles superiores del DOM.

```
<HTML>
<BODY>
  <DIV ID="capa1">
     <INPUT
        TYPE="checkbox"
        ID="check1"
     /> Confirmar
```

Curso de JavaScript

Manual Imprescindible

JavaScript lleva presente en el mundo Web prácticamente desde que Internet fue accesible para el mundo entero, pero ha sabido crecer y adaptarse a las nuevas necesidades y tendencias convirtiéndose en un lenguaje con un peso importante.

Este libro pretende enseñarle JavaScript desde su base para que pueda comprender perfectamente cómo está estructurado internamente y así le resulte mucho más sencillo dar el paso hacia los siguientes niveles, puesto que sus secretos no terminan en la última página.

La manera de abordar la materia será siempre de un modo progresivo, con explicaciones claras y mostrando uno o varios ejemplos de lo expuesto, de forma que un capítulo sirva como preludio del siguiente, como si se estuviera construyendo una escalera hacia su objetivo: dominar JavaScript.

2311350

ISBN 978-84-415-4228-0

ANAYA MULTIMEDIA

www.anayamultimedia.es