hello_

2. Fundamentos

COLECCIÓN FRONTEND

Javascript para Dummies

(porque programar no debería ser un enigma arcano)

Aprende desde cero cómo funciona JavaScript

Crea interacciones, lógica y magia real en el navegador

Domina variables, funciones, eventos y más

un día descubrí que addEventListener es como abrir un portal... a otro universo

@mihifidem

Desarrollador front-end, formador y defensor de que el undefined también merece amor

JavaScript

Fundamentos

Tabla de contenidos

JavaScript

Fundamentos

JavaScript es un lenguaje de programación interpretado y de alto nivel, diseñado para agregar interacción y dinamismo a las páginas web. Es compatible con todos los navegadores web modernos y se utiliza en una amplia variedad de aplicaciones, incluyendo desarrollo web front-end y back-end (con tecnologías como Node.js). JavaScript es uno de los lenguajes de programación más populares en el mundo, ofreciendo una amplia gama de características y bibliotecas para el desarrollo de aplicaciones web complejas.

JavaScript es un lenguaje de programación dinámico, orientado a objetos y basado en prototipos. Fue creado por Brendan Eich en Netscape en 1995 con el objetivo de agregar interactividad a las páginas web. Desde entonces, se ha convertido en uno de los lenguajes de programación más utilizados en el mundo, y es compatible con todos los navegadores web modernos.

JavaScript permite a los desarrolladores crear aplicaciones web dinámicas y interactivas, incluyendo características como formularios interactivos, juegos en línea, animaciones, gráficos, y mucho más. Además, JavaScript también es utilizado en aplicaciones de servidor con tecnologías como Node.js, lo que permite a los desarrolladores crear aplicaciones web completas que abarcan desde el lado del cliente hasta el lado del servidor.

El lenguaje de programación es fácil de aprender para aquellos con conocimientos previos en programación, y cuenta con una gran comunidad y una amplia gama de bibliotecas y herramientas disponibles en línea. Además, JavaScript es un lenguaje de programación versátil que se puede utilizar en una amplia variedad de aplicaciones, desde la creación de pequeñas aplicaciones web hasta la construcción de aplicaciones empresariales complejas.

Capítulo 1
Introducción a Javascript

1.1. Primer contacto

HTML y CSS incluyen instrucciones para indicar al navegador cómo debe organizar y visualizar un documento y su contenido, pero la interacción de estos lenguajes con el usuario y el sistema se limita solo a un grupo pequeño de respuestas predefinidas. Podemos crear un formulario con campos de entrada, controles y botones, pero HTML solo provee la funcionalidad necesaria para enviar la información introducida por el usuario al servidor o para limpiar el formulario. Algo similar pasa con CSS; podemos construir instrucciones (reglas) con seudoclases como :hover para aplicar un grupo diferente de propiedades cuando el usuario mueve el ratón sobre un elemento, pero si queremos realizar tareas personalizadas, como modificar los estilos de varios elementos al mismo tiempo, debemos cargar una nueva hoja de estilo que ya presente estos cambios. Con el propósito de alterar elementos de forma dinámica, realizar operaciones personalizadas, o responder al usuario y a cambios que ocurren en el documento, los navegadores incluyen un tercer lenguaje llamado JavaScript.

JavaScript es un lenguaje de programación que se usa para procesar información y manipular documentos. Al igual que cualquier otro lenguaje de programación, JavaScript provee instrucciones que se ejecutan de forma secuencial para indicarle al sistema lo que queremos que haga (realizar una operación aritmética, asignar un nuevo valor a un elemento, etc.). Cuando el navegador encuentra este tipo de código en nuestro documento, ejecuta las instrucciones al momento y cualquier cambio realizado en el documento se muestra en pantalla.

1.2. ¡Hola Mundo!

Esta parte del tutorial trata sobre el núcleo de JavaScript, el lenguaje en sí.

Pero necesitamos un entorno de trabajo para ejecutar nuestros scripts y el navegador es una buena opción. Mantendremos la cantidad de comandos específicos del navegador (como alert) al mínimo para que no pases tiempo en ellos si planeas concentrarte en otro entorno (como Node.js).

Primero, veamos cómo adjuntamos un script a una página web. Para entornos del lado del servidor (como Node.js), puedes ejecutar el script con un comando como "node my.js" .

La etiqueta “script”

Los programas de JavaScript se pueden insertar en casi cualquier parte de un documento HTML con el uso de la etiqueta *<script>***.**

```
<!DOCTYPE HTML>
<html>
<body>
   <p>Antes del script...</p>
   <script>
      alert( '¡Hola, mundo!' );
   </script>
   <p>...Después del script.</p>
</body>
</html>
```

La etiqueta *<script>* contiene código JavaScript que se ejecuta automáticamente cuando el navegador procesa la etiqueta.

La tradición de escribir "Hello, World!" como primer programa al aprender un nuevo lenguaje de programación proviene de un libro de texto influyente en ciencias de la computación. El libro "The C Programming Language", escrito por Brian Kernighan y Dennis Ritchie y publicado por primera vez en 1978, utilizó un programa "Hello, World!" como ejemplo inicial. Este programa simplemente muestra el mensaje "Hello, World!" en la pantalla.

La razón detrás de su popularidad es su simplicidad: el programa "Hello, World!" es lo suficientemente simple para que los principiantes lo entiendan y lo escriban, lo que les ayuda a aprender la estructura básica y la sintaxis de un nuevo lenguaje de programación. Al mismo tiempo, es lo suficientemente complejo como para involucrar varios aspectos fundamentales de la programación, como la salida de datos.

Desde entonces, "Hello, World!" se ha convertido en una especie de ritual de iniciación para programadores, utilizado en innumerables tutoriales y libros de texto como el primer paso para aprender programación. Su uso se ha extendido a casi todos los lenguajes de programación existentes.

1.3. Implementando JavaScript

Siguiendo el mismo enfoque que CSS, el código JavaScript se puede incorporar al documento mediante tres técnicas diferentes: el código se puede insertar en un elemento por medio de atributos (En línea), incorporar al documento como contenido del elemento <script> o cargar desde un archivo externo. La técnica En línea aprovecha atributos especiales que describen un evento, como un clic del ratón. Para lograr que un elemento responda a un evento usando esta técnica, todo lo que tenemos que hacer es agregar el atributo correspondiente con el código que queremos que se ejecute.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
</head>
<body>
  <section>
    <p onclick="alert('Hizo clic!')">Clic aquí</p>
    <p>No puede hacer clic aquí</p>
  </section>
</body>
</html>
```

Listado 1: *Definiendo JavaScript en línea*

El atributo onclick agregado al elemento <p> del Listado 1 dice algo similar a «cuando alguien hace clic en este elemento, ejecutar este código», y el código es (en este caso) la instrucción alert(). Esta es una instrucción predefinida en JavaScript llamada función. Lo que esta función hace es mostrar una ventana emergente con el valor provisto entre paréntesis. Cuando el usuario hace clic en el área ocupada por el elemento <p>, el navegador ejecuta la función alert() y muestra una ventana emergente en la pantalla con el mensaje «Hizo clic!».

Figura 1: *Ventana emergente generada por la función* `alert()`

Ejercicio 1: cree un nuevo archivo HTML con el código del Listado 1. Abra el documento en su navegador y haga clic en el texto «Clic Aquí»(el atributo **`onclick`** afecta a todo el elemento, no solo al texto, por lo que también puede hacer clic en el resto del área ocupada por el elemento para ejecutar el código). Debería ver una ventana emergente con el mensaje «Hizo clic!», tal como muestra la Figura 4-1.

Lo básico: JavaScript incluye múltiples funciones predefinidas y también permite crear funciones personalizadas. Estudiaremos cómo trabajar con funciones y funciones predefinidas más adelante en este capítulo.

El atributo onclick es parte de una serie de atributos provistos por HTML para responder a eventos. La lista de atributos disponibles es extensa, pero se pueden organizar en grupos dependiendo de sus propósitos. Por ejemplo, los siguientes son los atributos más usados asociados con el ratón.

onclick—Este atributo responde al evento **`click`**. El evento se ejecuta cuando el usuario hace clic con el botón izquierdo del ratón. HTML ofrece otros dos atributos similares llamados **`ondblclick`** (el usuario hace doble clic con el botón izquierdo del ratón) y **`oncontextmenu`** (el usuario hace clic con el botón derecho del ratón).

onmousedown—Este atributo responde al evento **`mousedown`**. Este evento se desencadena cuando el usuario pulsa el botón izquierdo o el botón derecho del ratón.

onmouseup—Este atributo responde al evento **`mouseup`**. El evento se desencadena cuando el usuario libera el botón izquierdo del ratón.

onmouseenter—Este atributo responde al evento **`mouseenter`**. Este evento se desencadena cuando el ratón se introduce en el área ocupada por el elemento.

onmouseleave—Este atributo responde al evento **`mouseleave`**. Este evento se desencadena cuando el ratón abandona el área ocupada por el elemento.

onmouseover—Este atributo responde al evento **`mouseover`**. Este evento se desencadena cuando el ratón se mueve sobre el elemento o cualquiera de sus elementos hijos.

onmouseout—Este atributo responde al evento **`mouseout`**. El evento se desencadena cuando el ratón abandona el área ocupada por el elemento o cualquiera de sus elementos hijos.

onmousemove—Este atributo responde al evento **`mousemove`**. Este evento se desencadena cada vez que el ratón se encuentra sobre el elemento y se mueve.

onwheel—Este atributo responde al evento **`wheel`**. Este evento se desencadena cada vez que se hace girar la rueda del ratón.

Los siguientes son los atributos disponibles para responder a eventos generados por el teclado. Estos tipos de atributos se aplican a elementos que aceptan una entrada del usuario, como los elementos <input> y <textarea>.

onkeypress—Este atributo responde al evento **`keypress`**. Este evento se desencadena cuando se activa el elemento y se pulsa una tecla.

onkeydown—Este atributo responde al evento **`keydown`**. Este evento se desencadena cuando se activa el elemento y se pulsa una tecla.

onkeyup—Este atributo responde al evento **`keyup`**. Este evento se desencadena cuando se activa el elemento y se libera una tecla.

También contamos con otros dos atributos importantes asociados al documento:

onload—Este atributo responde al evento **`load`**. El evento se desencadena cuando un recurso termina de cargarse.

onunload—Este atributo responde al evento **`unload`**. Este evento se desencadena cuando un recurso termina de cargarse.

Los atributos de evento se incluyen en un elemento dependiendo de cuándo queremos que se ejecute el código. Si queremos responder al clic del ratón, tenemos que incluir el atributo onclick, como hemos hecho en el Listado 4-1, pero si queremos iniciar un proceso cuando el puntero del ratón pasa sobre un elemento, tenemos que incluir los atributos onmouseover u onmousemove.

Debido a que en un elemento pueden ocurrir varios eventos en algunos casos al mismo tiempo, podemos declarar más de un atributo por cada elemento. Por ejemplo, el siguiente documento incluye un elemento <p> con dos atributos, onclick y onmouseout, que incluyen sus propios códigos JavaScript. Si el usuario hace clic en el elemento, se muestra una ventana emergente con el mensaje «Hizo clic!», pero si el usuario mueve el ratón fuera del área ocupada por el elemento, se muestra una ventana emergente diferente con el mensaje «No me abandone!».

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
</head>
<body>
  <section>
    <p onclick="alert('Hizo clic!')" onmouseout="alert('No me
abandone!')">Clic aquí</p>
  </section>
</body>
</html>
```

Listado 2: *Implementando múltiples atributos de evento*

Ejercicio 2 :actualice su archivo HTML con el código del Listado 2. Abra el documento en su navegador y mueva el ratón sobre el área ocupada por el elemento **<p>**. Si mueve el ratón fuera del área, debería ver una ventana emergente con el mensaje «No me abandone! ».

Los eventos no solo los produce el usuario, sino también el navegador. Un evento útil desencadenado por el navegador es load. Este evento se desencadena cuando se ha terminado de cargar un recurso y, por lo tanto, se utiliza frecuentemente para ejecutar código JavaScript después de que el navegador ha cargado el documento y su contenido.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
</head>
<body onload="alert('Bienvenido!')">
  <section>
    <h1>Mi Sitio Web</h1>
    <p>Bienvenido a mi sitio web</p>
  </section>
</body>
</html>
```

Listado 3: *Respondiendo al evento* `load`

El documento del Listado 3 muestra una ventana emergente para dar la bienvenida al usuario después de que se ha cargado completamente. El navegador primero carga el contenido del documento y cuando termina, llama a la función alert() y muestra el mensaje en la pantalla.

IMPORTANTE: los eventos son críticos en el desarrollo web. Además de los estudiados en este capítulo, hay docenas de eventos disponibles para controlar una variedad de procesos, desde reproducir un vídeo hasta controlar el progreso de una tarea. Estudiaremos eventos más adelante en este capítulo e introduciremos el resto de los eventos disponibles en situaciones más prácticas en capítulos posteriores.

Lo básico: cuando pruebe el código del Listado 3 en su navegador, verá quela ventana emergente se muestra antes de que el contenido del documento aparezca en la pantalla. Esto se debe a que el documento se carga en una estructura interna de objetos llamada DOM y luego se reconstruye en la pantalla desde estos objetos. Estudiaremos la estructura DOM y cómo acceder a los elementos HTML desde JavaScript más adelante en este capítulo.

Los atributos de evento son útiles cuando queremos probar código o implementar una función de inmediato, pero no son apropiados para aplicaciones importantes. Para trabajar con códigos extensos y personalizar las funciones, tenemos que agrupar el código con el elemento <script>. El elemento <script> actúa igual que el elemento <style> para CSS, organizando el código en un solo lugar y afectando al resto de los elementos en el documento usando referencias.

```
<!DOCTYPE html>
 <html lang="es">
 <head>
   <meta charset="utf-8">
   <title>JavaScript</title>
   <script>
     alert('Bienvenido!');
   </script>
 </head>
 <body>
   <section>
     <p>Hola</p>
   </section>
 </body>
 </html>
```

Listado 4: *Código JavaScript introducido en el documento*

El elemento <script> y su contenido se pueden ubicar en cualquier parte del documento, pero normalmente se introducen dentro de la cabecera, como hemos hecho en este ejemplo. De esta manera, cuando el navegador carga el archivo, lee el contenido del elemento <script>, ejecuta el código al instante, y luego continúa procesando el resto del documento.

Ejercicio 3:actualice su archivo HTML con el código del Listado 4 y abra el documento en su navegador. Debería ver una ventana emergente con el mensaje «Bienvenido!» tan pronto como se carga el documento. Debido a que la función **`alert()`** detiene la ejecución del código, el contenido del documento no se muestra en la pantalla hasta que pulsamosel botón OK.

Introducir JavaScript en el documento con el elemento <script> puede resultar práctico cuando tenemos un grupo pequeño de instrucciones, pero el código JavaScript crece con rapidez en aplicaciones profesionales. Si usamos el mismo código en más de un documento, tendremos que mantener diferentes versiones del mismo programa y los navegadores tendrán que descargar el mismo código una y otra vez con cada documento solicitado por el usuario. Una alternativa es introducir el código JavaScript en un archivo externo y luego cargarlo desde los documentos que lo requieren. De esta manera, solo los documentos que necesitan ese grupo de instrucciones deberán incluir el archivo, y el navegador tendrá que descargar el archivo una sola vez (los navegadores mantienen los archivos en un caché en el ordenador del usuario en caso de que sean requeridos más adelante por otros documentos del mismo sitio web). Para este propósito, el elemento <script> incluye el atributo src. Con este atributo, podemos declarar la ruta al archivo JavaScript y escribir todo nuestro código dentro de este archivo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script src="micodigo.js"></script>
</head>
<body>
  <section>
    <p>Hola</p>
  </section>
</body>
</html>
```

Listado 5: Introduciendo código JavaScript desde un archivo externo

El elemento <script> del Listado 5 carga el código JavaScript desde una archivo llamado micodigo.js. A partir de ahora, podemos insertar este archivo en cada documento de nuestro sitio web y reusar el código cada vez que lo necesitemos.

Al igual que los archivos HTML y CSS, los archivos JavaScript son simplemente archivos de texto que podemos crear con cualquier editor de texto o los editores profesionales que recomendamos en el Capítulo 1, como Atom (www.atom.io). A estos tipos de archivos se les puede asignar cualquier nombre, pero por convención tienen que tener la extensión .js. El archivo debe contener el código JavaScript exactamente según se declara entre las etiquetas <script>. Por ejemplo, el siguiente es el código que tenemos que introducir en el archivo micodigo.js para reproducir el ejemplo anterior.

```
alert("Bienvenido!");
```

Listado 6: Creando un archivo JavaScript (micodigo.js)

Ejercicio 4: actualice su archivo HTML con el código del Listado 5. Cree un nuevo archivo con el nombre micodigo.js y el código JavaScript del Listado 6. Abra el documento en su navegador. Debería ver una ventana emergente con el mensaje «Bienvenido!» tan pronto como se carga el documento.

Lo básico: en JavaScript se recomienda finalizar cada instrucción con un punto y coma para asegurarnos de que el navegador no tenga ninguna dificultad al identificar el final de cada instrucción. El punto y coma se puede ignorar, pero nos puede ayudar a evitar errores cuando el código está compuesto por múltiples instrucciones, como ocurre frecuentemente.

IMPORTANTE: además del atributo `src`, el elemento `<script>` también puede incluir los atributos `async` y `defer`. Estos son atributos booleanos que indican cómo y cuándo se debe ejecutar el código. Si el atributo `async` se declara, el código se ejecuta de forma asíncrona (mientras se procesa el resto del documento). Si el atributo `defer` se declara en su lugar, el código se ejecuta después de que el documento completo se haya procesado.

1.4. Estructura del código

Lo primero que estudiaremos son los bloques de construcción del código.

Sentencias

Las sentencias son construcciones sintácticas y comandos que realizan acciones.
Ya hemos visto una sentencia, alert('¡Hola mundo!') , que muestra el mensaje —¡Hola mundo!.

Podemos tener tantas sentencias en nuestro código como queramos, las cuales se pueden separar con un punto y coma.

ejemplo, aquí separamos —Hello World en dos alerts:

```
alert('Hola'); alert('Mundo');
```

Listado 7:

Generalmente, las sentencias se escriben en líneas separadas para hacer que el código sea más legible:

```
alert('Hola');
alert('Mundo');
```

Listado 8:

Punto y coma

Se puede omitir un punto y coma en la mayoría de los casos cuando existe un salto de línea. Esto también funcionaría:

```
lert('Hola')
alert('Mundo')
```

Listado 9:

Aquí, JavaScript interpreta el salto de línea como un punto y coma —implícito Esto se denomina inserción automática de punto y coma .

En la mayoría de los casos, una nueva línea implica un punto y coma. Pero “en la mayoría de los casos” no significa “siempre”!

Hay casos en que una nueva línea no significa un punto y coma. Por ejemplo:

```
3 +
1
+ 2);
```

Listado 9:

El código da como resultado *6* porque JavaScript no inserta punto y coma aquí. Es intuitivamente obvio que si la línea termina con un signo más "+" , es una —expresión incompleta un punto y coma aquí sería incorrecto. Y en este caso eso funciona según loprevisto.

Pero hay situaciones en las que JavaScript "falla" al asumir un punto y coma donde realmente se necesita.

Los errores que ocurren en tales casos son bastante difíciles de encontrar y corregir.

Un ejemplo de error

Si tienes curiosidad por ver un ejemplo concreto de tal error, mira este código:

```
alert("Hello");

[1, 2].forEach(alert);
```

Quitemos el punto y coma del alert:

```
alert("Hello")

[1, 2].forEach(alert);
```

La diferencia, comparando con el código anterior, es de solo un carácter: falta el punto y coma al final de la primera línea.

Esta vez, si ejecutamos el código, solo se ve el primer Hello (y un error pero necesitas abrir la consola para verlo). Los números no aparecen más.

Esto ocurre porque JavaScript no asume un punto y coma antes de los corchetes [...] , entonces el código del primer ejemplo se trata como una sola sentencia.

Así es como lo ve el motor:

```
alert("Hello")[1, 2].forEach(alert);
```

Se ve extraño, ¿verdad? Tal unión en este caso es simplemente incorrecta. Necesitamos poner un punto y coma después del alert para que el código funcione bien.

Recomendamos colocar puntos y coma entre las sentencias, incluso si están separadas por saltos de línea. Esta regla está ampliamente adoptada por la comunidad. Notemos una vez más que es posible omitir los puntos y coma la mayoría del tiempo. Pero es más seguro, especialmente para un principiante, usarlos.

Comentarios

A medida que pasa el tiempo, los programas se vuelven cada vez más complejos. Se hace necesario agregar comentarios que describan lo que hace el código y por qué.
Los comentarios se pueden poner en cualquier lugar de un script. No afectan su ejecución porque el motor simplemente los ignora.

Los comentarios de una línea comienzan con dos caracteres de barra diagonal // .

El resto de la línea es un comentario. Puede ocupar una línea completa propia o seguir una sentencia.

Como aquí:

```
// Este comentario ocupa una línea propia.
alert('Hello');

alert('World'); // Este comentario sigue a la sentencia.
```

Los comentarios de varias líneas comienzan con una barra inclinada y un asterisco /* y terminan con un asterisco y una barra inclinada */ .

Como aquí:

```
/* Un ejemplo con dos mensajes.
Este es un comentario multilínea.
*/
alert('Hola');
alert('Mundo');
```

El contenido de los comentarios se ignora, por lo que si colocamos el código dentro de */ , no se ejecutará.

A veces puede ser útil deshabilitar temporalmente una parte del código:

```
/* Comentando el código
alert('Hola');
*/
alert('Mundo');
```

¡Usa accesos rápidos del teclado!

En la mayoría de los editores, se puede comentar una línea de código presionando `Ctrl+/` para un comentario de una sola línea y algo como `Ctrl+Shift+/` – para comentarios devarias líneas (selecciona una parte del código y pulsa la tecla de acceso rápido). Para Mac, intenta `Cmd` en lugar de `Ctrl` y `Option` en lugar de

Por favor, no dudes en comentar tu código.

```
let message = "This";

// 'let' repetidos lleva a un error
let message = "That"; // SyntaxError: 'message' ya fue declarado
```

Los aumentan el tamaño general del código, pero eso no es un problema en absoluto. Hay muchas herramientas que minimizan el código antes de publicarlo en un servidor de producción. Eliminan los , por lo que no aparecen en los scripts de trabajo. Por lotanto, los comentarios no tienen ningún efecto negativo en la producción.

Más adelante, en el tutorial, habrá un capítulo Estilo de codificación que también explica cómoescribir mejores comentarios.

El modo moderno, "use strict"

Durante mucho tiempo, JavaScript evolucionó sin problemas de compatibilidad. Se añadíannuevas características al lenguaje sin que la funcionalidad existente cambiase.

Esto tenía el beneficio de nunca romper código existente, pero lo malo era que cualquier error odecisión incorrecta tomada por los creadores de JavaScript se quedaba para siempre en el lenguaje.

Esto fue así hasta 2009, cuando ECMAScript 5 (ES5) apareció. Esta versión añadió nuevascaracterísticas al lenguaje y modificó algunas de las ya existentes.

Para mantener el códigoantiguo funcionando, la mayor parte de las modificaciones están desactivadas por defecto. Tienes que activarlas explícitamente usando una directiva especial: `"use strict"` .

“use strict”

La directiva se asemeja a un string: `"use strict"` . Cuando se sitúa al principio de un script,el script entero funciona de la manera “moderna”.

Por ejemplo:

```
"use strict";
// este código funciona de la manera moderna
...
```

Listado 10

¿Deberíamos utilizar “use strict”?

La pregunta podría parecer obvia, pero no lo es.

Uno podría recomendar que se comiencen los script con "use strict" … ¿Pero sabes lo que es interesante?

El JavaScript moderno admite “clases” y “módulos”, estructuras de lenguaje avanzadas (que seguramente llegaremos a ver), que automáticamente habilitan use strict . Entonces no necesitamos agregar la directiva

Entonces, por ahora si las usamos; es un invitado bienvenido al tope de tus scripts.

Luego, cuando tu código sea todo clases y módulos, puedes omitirlo.

A partir de ahora tenemos que saber acerca de use strict en general.

1.5. Variables

Una analogía de la vida real

Podemos comprender fácilmente el concepto de una "variable" si nos la imaginamos como una "caja" con una etiqueta de nombre único pegada en ella.

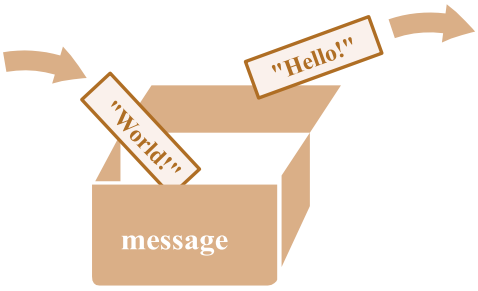
Por ejemplo, podemos imaginar la variable *message* como una caja etiquetada *"message"* con el valor *"hola"* adentro:

Podemos introducir cualquier valor a la caja.
También la podemos cambiar cuantas veces queramos:

```
let message;
message =
'Hola!';
message = 'Mundo!'; // valor alterado

alert(message);
```

Cuando el valor ha sido alterado, los datos antiguos serán removidos de la variable:



También podemos declarar dos variables y copiar datos de una a la otra.

```
let hello = 'Hola mundo!';

let message;

// copia 'Hola mundo' de hello a message
message = hello;

// Ahora, ambas variables contienen los mismos
datosalert(hello); // Hola mundo!
alert(message); // Hola mundo!
```

Importante!! Declarar dos veces lanza un error
Una variable debe ser declarada solamente una vez.
Una declaración repetida de la misma variable es un error:

```
let message = "This";

// 'let' repetidos lleva a un error
let message = "That"; // SyntaxError: 'message' ya fue declarado
```

Nombramiento de variables

Existen dos limitaciones de nombre de variables en JavaScript:

1. El nombre únicamente puede incluir letras, dígitos, o los símbolos
2. El primer carácter no puede ser un dígito.

Ejemplos de nombres válidos:

```
let userName;
let test123;
```

Cuando el nombre contiene varias palabras, se suele usar el estilo camelCase (capitalización en camello), donde las palabras van pegadas una detrás de otra, con cada inicial en mayúscula: miNombreEsMuyLargo

Es interesante notar que el símbolo del dólar “$” y el guion bajo “_” también se utilizan en nombres. Son símbolos comunes, tal como las letras, sin ningún significado especial.

Los siguientes nombres son válidos:

```
let $ = 1; // Declara una variable con el nombre "$"
let _ = 2; // y ahora una variable con el nombre "_"

alert($ + _); // 3
```

Ejemplos de nombres incorrectos:

```
let la; // no puede iniciar con un dígito

let my-name; // los guiones '-' no son permitidos en nombres
```

Por supuesto, JavaScript es algo más que ventanas emergentes mostrando mensajes para alertar al usuario. El lenguaje puede realizar numerosas tareas, desde calcular algoritmos complejos hasta procesar el contenido de un documento. Cada una de estas tareas involucra la manipulación de valores, y esta es la razón por la que la característica más importante de JavaScript, al igual que cualquier otro lenguaje de programación, es la capacidad de almacenar datos en memoria.

La memoria de un ordenador o dispositivo móvil es como un panal de abejas con millones y millones de celdas consecutivas en las que se almacena información. Estas celdas tienen un espacio limitado y, por lo tanto, es necesaria la combinación de múltiples celdas para almacenar grandes cantidades de datos. Debido a la complejidad de esta estructura, los lenguajes de programación incorporan el concepto de variables para facilitar la identificación de cada valor almacenado en memoria. Las variables son simplemente nombres asignados a una celda o un grupo de celdas donde se van a almacenar los datos.

Por ejemplo, si almacenamos el valor 5, tenemos que saber en qué parte de la memoria se encuentra para poder leerlo más adelante. La creación de una variable nos permite identificar ese espacio de memoria con un nombre y usar ese nombre más adelante para leer el valor o reemplazarlo por otro.

```
let message = "This";

// 'let' repetidos lleva a un error
let message = "That"; // SyntaxError: 'message' ya fue declarado
```

Las variables en JavaScript se declaran con la palabra clave var seguida del nombre que queremos asignarle. Si queremos almacenar un valor en el espacio de memoria asignado por el sistema a la variable, tenemos que incluir el carácter = (igual) seguido del valor, como en el siguiente ejemplo.

```
var minumero = 2;
```

Listado 1.5-1: *Declarando una variable en JavaScript*

La instrucción del Listado 1.5-1 crea la variable minumero y almacena el valor 2 en el espacio de memoria reservado por el sistema para la misma. Cuando se ejectua este código, el navegador reserva un espacio en memoria, almacena el número 2 en su interior, crea una referencia a ese espacio, y finalmente asigna esta referencia al nombre minumero.

Después de asignar el valor a la variable, cada vez que se referencia esta variable (se usa el nombre minumero), el sistema lee la memoria y devuelve el número 2, tal como ilustra el siguiente ejemplo.

```
var minumero = 2;
alert(minumero);
```

Listado-1.5-2: *Usando el contenido de una variable*

Como ya hemos mencionado, las instrucciones en un programa JavaScript las ejecuta el navegador una por una en secuencia. Por lo tanto, cuando el navegador lee el código del Listado 1.5-2, ejecuta las instrucciones de arriba abajo. La primera instrucción le pide al navegador que cree una variable llamada minumero y le asigne el valor 2. Después de completar esta tarea, el navegador ejecuta la siguiente instrucción de la lista. Esta instrucción le pide al navegador que muestre una ventana emergente con el valor actual almacenado en la variable minumero.

Figura 1.5-1: *Ventana emergente mostrando el valor de una variable*

Ejercicio 5 :actualice su archivo micodigo.js con el código del Listado 1.5-2. Abra el documento del Listado 5 en su navegador. Debería ver una ventana emergente con el valor **2**.

Las variables se denominan así porque sus valores no son constantes. Podemos cambiar sus valores cada vez que lo necesitemos, y esa es, de hecho, su característica más importante.

```
var minumero = 2;
minumero = 3;
alert(minumero);
```

Listado 1.5-3: *Asignando un nuevo valor a la variable*

En el Listado 1.5-3, después de que se declara la variable, le es asignado un nuevo valor. Ahora, la función alert() muestra el número 3 (la segunda instrucción reemplaza el valor 2 por el valor 3). Cuando asignamos un nuevo valor a una variable, no tenemos la necesidad de declarar la palabra clave var, solo se requieren el nombre de la variable y el carácter =.

El valor almacenado en una variable se puede asignar a otra. Por ejemplo, el siguiente código crea dos variables llamadas minumero y tunumero, y asigna el valor almacenado en la variable minumero a la variable tunumero. El valor mostrado en la pantalla es el número 2.

```
var minumero = 2;
var tunumero = minumero;
alert(tunumero);
```

Listado 1.5-4: *Asignando el valor de una variable a otra variable*

En una situación más práctica, probablemente usaríamos el valor de la variable para ejecutar una operación y asignar el resultado de vuelta a la misma variable.

```
var minumero = 2;
minumero = minumero + 1;   // 3
alert(minumero);
```

Listado 1.5-5: *Realizando una operación con el valor almacenado en una variable*

En este ejemplo, el valor 1 se agrega al valor actual de minumero y el resultado se asigna a la misma variable. Esto es lo mismo que sumar 2 + 1, con la diferencia de que cuando usamos una variable en lugar de un número, su valor puede cambiar en cualquier momento.

Lo básico: los caracteres al final de la segunda instrucción se consideran comentarios y, por lo tanto, no se procesan como parte de la instrucción. Los comentarios se pueden agregar al código como referencias o recordatorios para el desarrollador. Se pueden declarar usando dos barras oblicuas para comentarios de una línea (**`// comentario`**) o combinando una barra oblicua con un asterisco para crear comentarios de varias líneas (**`/* comentario */`**). Todo lo que se encuentra a continuación de las dos barras o entre los caracteres **`/*`** y **`*/`** el navegador lo ignora.

Además del operador +, JavaScript también incluye los operadores - (resta), * (multiplicación), / (división), y % (módulo). Estos operadores se pueden usar en una operación sencilla entre dos valores o combinados con múltiples valores para realizar operaciones aritméticas más complejas.

```
var minumero = 2;
minumero = minumero * 25 + 3;   // 53
alert(minumero);
```

Listado 1.5-6: *Realizando operaciones complejas*

Las operaciones aritméticas se ejecutan siguiendo un orden de prioridad determinado por los operadores. La multiplicación y la división tienen prioridad sobre la adición y la substracción. Esto significa que las multiplicaciones y divisiones se calcularán antes que las sumas y restas. En el ejemplo del Listado 1.5-6, el valor actual de la variable minumero se multiplica por 25, y luego el valor 3 se suma al resultado. Si queremos controlar la precedencia, podemos aislar las operaciones con paréntesis. Por ejemplo, el siguiente código realiza la adición primero y luego la multiplicación, lo que genera un resultado diferente.

```
var minumero = 2;
minumero = minumero * (25 + 3);   // 56
alert(minumero);
```

Listado 1.5-7 *Controlando la precedencia en la operación*

Realizar una operación en el valor actual de una variable y asignar el resultado de vuelta a la misma variable es muy común en programación. JavaScript ofrece los siguientes operadores para simplificar esta tarea.

- **++** es una abreviatura de la operación **`variable = variable + 1`**.
- **--** es una abreviatura de la operación **`variable = variable - 1`**.
- **+=** es una abreviatura de la operación **`variable = variable + number`**.
- **-=** es una abreviatura de la operación **`variable = variable - number`**.
- ***=** es una abreviatura de la operación **`variable = variable * number`**.
- **/=** es una abreviatura de la operación **`variable = variable / number`**.

Con estos operadores, podemos realizar operaciones en los valores de una variable y asignar el resultado de vuelta a la misma variable. Un uso común de estos operadores es el de crear contadores que incrementan o disminuyen el valor de una variable en una o más unidades. Por ejemplo, el operador ++ suma el valor 1 al valor actual de la variable cada vez que se ejecuta la instrucción.

```
var minumero = 0;
minumero++;
alert(minumero);  // 1
```

Listado 1.5-8: *Incrementando el valor de una variable*

Si el valor de la variable se debe incrementar más de una unidad, podemos usar el operador +=. Este operador suma el valor especificado en la instrucción al valor actual de la variable y almacena el resultado de vuelta en la misma variable.

```
var minumero = 0;
minumero += 5;
alert(minumero);  // 5
```

Listado 1.5-9: *Incrementando el valor de una variable en un valor específico*

El proceso generado por el código del Listado 1.5-9 es sencillo. Después de asignar el valor 0 a la variable minumero, el sistema lee la segunda instrucción, obtiene el valor actual de la variable, le suma el valor 5 y almacena el resultado de vuelta en minumero.

Una operación interesante que aún no hemos implementado es el operador módulo. Este operador devuelve el resto de una división entre dos números.

```
var minumero = 11 % 3;  // 2
alert(minumero);
```

Listado 1.5-20: *Calculando el resto de una división*

La operación asignada a la variable minumero del Listado 1.5-20 produce el resultado 2. El sistema divide 11 por 3 y encuentra el cociente 3. Luego, para obtener el resto, calcula 11 menos la multiplicación de 3 por el cociente (11 - (3 * 3) = 2).

El operador módulo se usa frecuentemente para determinar si un valor es par o impar. Si calculamos el resto de un número entero dividido por 2, obtenemos un resultado que indica si el número es par o impar. Si el número es par, el resto es 0, pero si el número es impar, el resto es 1 (o -1 para valores negativos).

```
var minumero = 11;
alert(minumero % 2);   // 1
```

Listado 1.5-21: *Determinando la paridad de un número*

El código del Listado 1.5-21 calcula el resto del valor actual de la variable minumero dividido por 2. El valor que devuelve es 1, lo que significa que el valor de la variable es impar.

En este ejemplo ejecutamos la operación entre los paréntesis de la función alert(). Cada vez que una operación se incluye dentro de una instrucción, el navegador primero calcula la operación y luego ejecuta la instrucción con el resultado, por lo que una operación puede ser provista cada vez que se requiere un valor.

Ejercicio 6:actualice su archivo micodigo.js con el ejemplo que quiere probar y abra el documento en su navegador. Reemplace los valores y realice operaciones más complejas para ver los diferentes resultados producidos por JavaScript y así familiarizarse con los operadores del lenguaje.

1.6. Tipo de variables

Cadenas de texto

En los anteriores ejemplos hemos almacenado números, pero las variables también se pueden usar para almacenar otros tipos de valores, incluido texto. Para asignar texto a una variable, tenemos que declararlo entre comillas simples o dobles.

```
var mitexto = "Hola Mundo!";
alert(mitexto);
```

Listado : *Asignando una cadena de caracteres a una variable*

El código del Listado crea una variable llamada mitexto y le asigna una cadena de caracteres. Cuando se ejecuta la primera instrucción, el sistema reserva un espacio de memoria lo suficientemente grande como para almacenar la cadena de caracteres, crea la variable, y almacena el texto. Cada vez que leemos la variable mitexto, recibimos en respuesta el texto «Hola Mundo!» (sin las comillas).

Figura : *Ventana emergente mostrando una cadena de caracteres*

El espacio reservado en memoria por el sistema para almacenar la cadena de caracteres depende del tamaño del texto (cuántos caracteres contiene), pero el sistema está preparado para ampliar este espacio si luego se asignan valores más extensos a la variable. Una situación común en la cual se asignan textos más extensos a la misma variable es cuando agregamos más caracteres al comienzo o al final del valor actual de la variable. El texto se puede concatenar con el operador +.

```
var mitexto = "Mi nombre es ";
mitexto = mitexto + "Juan";
alert(mitexto);
```

Listado : *Concatenando texto*

El código del Listado agrega el texto "Juan" al final del texto "Mi nombre es ". El valor final de la variable mitexto es "Mi nombre es Juan". Si queremos agregar el texto al comienzo, solo tenemos que invertir la operación.

```
var mitexto = "Juan";
mitexto = "Mi nombre es " + mitexto;
alert(mitexto);
```

Listado : *Agregando texto al comienzo del valor*

Si en lugar de texto intentamos concatenar una cadena de caracteres con un número, el número se convierte en una cadena de caracteres y se agrega al valor actual. El siguiente código produce la cadena de caracteres "El número es 3".

```
var mitexto = "El número es " + 3;
alert(mitexto);
```

Listado : *Concatenando texto con números*

Este procedimiento es importante cuando tenemos una cadena de caracteres que contiene un número y queremos agregarle otro número. Debido a que JavaScript considera el valor actual como una cadena de caracteres, el número también se convierte en una cadena de caracteres y los valores no se suman.

```
var mitexto = "20" + 3;
alert(mitexto);  // "203"
```

Listado *Concatenando números*

Lo básico: el resultado del código del Listado no es 23, sino la cadena de caracteres "203". El sistema convierte el número 3 en una cadena de caracteres y concatena las cadenas en lugar de sumar los números. Más adelante aprenderemos cómo extraer números de una cadena de caracteres para poder realizar operaciones aritméticas con estos valores.

Las cadenas de caracteres pueden contener cualquier carácter que queramos, y esto incluye comillas simples o dobles. Si las comillas en el texto son diferentes a las comillas usadas para definir la cadena de caracteres, estas se tratan como cualquier otro carácter, pero si las comillas son las mismas, el sistema no sabe dónde termina el texto. Para resolver este problema, JavaScript ofrece el carácter de escape \. Por ejemplo, si la cadena de caracteres se ha declarado con comillas simples, tenemos que escapar las comillas simples dentro del texto.

```
var mitexto = 'El \'libro\' es interesante';
alert(mitexto);  // "El 'libro' es interesante"
```

Listado : *Escapando caracteres*

JavaScript ofrece varios caracteres de escape con diferentes propósitos. Los que se utilizan con más frecuencia son \n para generar una nueva línea y \r para devolver el cursor al comienzo de la línea. Generalmente, estos dos caracteres se implementan en conjunto para dividir el texto en múltiples líneas, tal como muestra el siguiente ejemplo.

```
var mitexto = "Felicidad no es hacer lo que uno quiere\r\n";
mitexto = mitexto + "sino querer lo que uno hace."
alert(mitexto);
```

Listado : *Generando nuevas líneas de texto*

El código del Listado comienza asignando una cadena de caracteres a la variable mitexto que incluye los caracteres de escape \r\n. En la segunda instrucción, agregamos otro texto al final del valor actual de la variable, pero debido a los caracteres de escape, estos dos textos se muestran dentro de la ventana emergente en diferentes líneas.

Booleanos

Otro tipo de valores que podemos almacenar en variables son los booleanos. Las variables booleanas pueden contener solo dos valores: true (verdadero) o false (falso). Estas variables son particularmente útiles cuando solo necesitamos determinar el estado actual de una condición. Por ejemplo, si nuestra aplicación necesita saber si un valor insertado en el formulario es válido o no, podemos informar de esta condición al resto del código con una variable booleana.

```
var valido = true;
alert(valido);
```

Listado : *Declarando una variable booleana*

El propósito de estas variables es el de simplificar el proceso de identificación del estado de una condición. Si usamos un número entero para indicar un estado, deberemos recordar qué números decidimos usar para representar los estados válido y no válido. Usando valores booleanos en su lugar, solo tenemos que comprobar si el valor es igual a true o false.

IMPORTANTE: los valores booleanos son útiles cuando los usamos junto con instrucciones que nos permiten realizar una tarea o tareas repetitivas de acuerdo a una condición. Estudiaremos las condicionales y los bucles más adelante en este capítulo.

Arrays

Las variables también pueden almacenar varios valores al mismo tiempo en una estructura llamada array. Los arrays se pueden crear usando una sintaxis simple que incluye los valores separados por comas dentro de corchetes. Los valores se identifican luego mediante un índice, comenzando desde 0 (cero).

```
var miarray = ["rojo", "verde", "azul"];
alert(miarray[0]);  // "rojo"
```

Listado : *Creando arrays*

En el Listado , creamos un array llamado miarray con tres valores, las cadenas de caracteres "rojo", "verde" y "azul". JavaScript asigna automáticamente el índice 0 al primer valor, 1 al segundo, y 2 al tercero. Para leer estos datos, tenemos que mencionar el índice del valor entre corchetes después del nombre de la variable. Por ejemplo, para obtener el primer valor de miarray, tenemos que escribir la instrucción miarray[0], como hemos hecho en nuestro ejemplo.

La función alert() puede mostrar no solo valores independientes, sino arrays completos. Si queremos ver todos los valores incluidos en el array, solo tenemos que especificar el nombre del array.

```
var miarray = ["rojo", "verde", "azul"];
alert(miarray);  // "rojo,verde,azul"
```

Listado : *Mostrando los valores del array*

Los arrays, al igual que cualquier otra variable, pueden contener cualquier tipo de valor que deseemos. Por ejemplo, podemos crear un array como el del Listado combinando números y cadenas de caracteres.

```
var miarray = ["rojo", 32, "HTML5 es genial!"];
alert(miarray[1]);
```

Listado: *Almacenando diferentes tipos de valores*

Ejercicio 7:reemplace el código en su archivo micodigo.js por el código del Listado 4-28 y abra el documento del Listado 4-5 en su navegador.Cambie el índice provisto en la función **`alert()`** para mostrar cada valor en el array (recuerde que los índices comienzan desde 0).

Si intentamos leer un valor en un índice que aún no se ha definido, JavaScript devuelve el valor undefined (indefinido) Este valor lo usa el sistema para informar de que el valor que estamos intentando acceder no existe, pero también podemos asignarlo a un array cuando aún no contamos con el valor para esa posición.

```
var miarray = ["rojo", undefined, 32];
alert(miarray[1]);
```

Listado : *Declarando valores indefinidos*

Otra manera mejor de indicarle al sistema que no existe un valor disponible en un momento para un índice del array es usando otro valor especial llamado null (nulo). La diferencia entre los valores undefined y null es que undefined indica que la variable fue declarada pero ningún valor le fue asignado, mientras que null indica que existe un valor, pero es nulo.

```
var miarray = ["rojo", 32, null];
alert(miarray[2]);
```

Listado 4-30: *Declarando el valor* `null`

Por supuesto, también podemos realizar operaciones en los valores de un array y almacenar los resultados, como hemos hecho antes con variables sencillas.

```
var miarray = [64, 32];
miarray[1] = miarray[1] + 10;
alert("El valor actual es " + miarray[1]); // "El valor actual es 42"
```

Listado : *Trabajando con los valores del array*

Los arrays trabajan exactamente igual que otras variables, con la excepción de que tenemos que mencionar el índice cada vez que queremos usarlos. Con la instrucción miarray[1] = miarray[1] + 10 le decimos al intérprete de JavaScript que lea el valor actual de miarray en el índice 1 (32), le sume 10, y almacene el resultado en el mismo array e índice; por lo que al final el valor de miarray[1] es 42.

Los arrays pueden incluir cualquier tipo de valores, por lo que es posible declarar arrays de arrays. Estos tipos de arrays se denominan arrays multidimensionales.

```
var miarray = [[2, 45, 31], [5, 10], [81, 12]];
```

Listado : *Definiendo arrays multidimensionales*

El ejemplo del Listado crea un array de arrays de números enteros. Para acceder a estos valores, tenemos que declarar los índices de cada nivel entre corchetes, uno después del otro. El siguiente ejemplo devuelve el primer valor (índice 0) del segundo array (índice 1).

La instrucción busca el array en el índice 1 y luego busca por el número en el índice 0.

```
var miarray = [[2, 45, 31], [5, 10], [81, 12]];
alert(miarray[1][0]);  // 5
```

Listado : *Accediendo a los valores en arrays multidimensionales*

Si queremos eliminar uno de los valores, podemos declararlo como undefined o null, como hemos hecho anteriormente, o declararlo como un array vacío asignando corchetes sin valores en su interior.

```
var miarray = [[2, 45, 31], [5, 10], [81, 12]];
miarray[1] = []
alert(miarray[1][0]);  // undefined
```

Listado : *Asignando un array vacío como el valor de otro array*

Ahora, el valor mostrado en la ventana emergente es undefined, porque no hay ningún elemento en las posición [1][0]. Por supuesto, esto también se puede usar para vaciar cualquier tipo de array.

```
var miarray = [2, 45, 31];
miarray = []
alert(miarray[1]);  // undefined
```

Listado : *Asignando un array vacío a una variable*

Lo básico: al igual que las cadenas de caracteres, los arrays también se declaran como objetos en JavaScript y, por lo tanto, incluyen métodos para realizar operaciones en sus valores. Estudiaremos objetos, los objetos **`array`**, y cómo implementar sus métodos más adelante en este capítulo.

1.7. Condicionales y bucles

Hasta este punto hemos escrito instrucciones en secuencia, una debajo de la otra. En este tipo de programas, el sistema ejecuta cada instrucción una sola vez y en el orden en el que se presentan. Comienza con la primera y sigue hasta llegar al final de la lista. El propósito de condicionales y bucles es el de romper esta secuencia. Los condicionales nos permiten ejecutar una o más instrucciones solo cuando se cumple una determinada condición, y los bucles nos permiten ejecutar un bloque de código (un grupo de instrucciones) una y otra vez hasta que se satisface una condición. JavaScript ofrece un total de cuatro instrucciones para procesar código de acuerdo a condiciones determinadas por el programador: if, switch, for y while.

La manera más simple de comprobar una condición es con la instrucción if. Esta instrucción analiza una expresión y procesa un grupo de instrucciones si la condición establecida por esa expresión es verdadera. La instrucción requiere la palabra clave if seguida de la condición entre paréntesis y las instrucciones que queremos ejecutar si la condición es verdadera entre llaves.

```
var mivariable = 9;
if (mivariable < 10) {
  alert("El número es menor que 10");
}
```

Listado 4-36: *Comprobando una condición con* `if`

En el código del Listado 4-36, el valor 9 se asigna a mivariable, y luego, usando if comparamos la variable con el número 10. Si el valor de la variable es menor que 10, la función alert() muestra un mensaje en la pantalla.

El operador usado para comparar el valor de la variable con el número 10 se llama operador de comparación. Los siguientes son los operadores de comparación disponibles en JavaScript.

- **==** comprueba si el valor de la izquierda es igual al de la derecha.
- **!=** comprueba si el valor de la izquierda es diferente al de la derecha.
- **>** comprueba si el valor de la izquierda es mayor que el de la derecha.
- **<** comprueba si el valor de la izquierda es menor que el de la derecha.
- **>=** comprueba si el valor de la izquierda es mayor o igual que el de la derecha.
- **<=** comprueba si el valor de la izquierda es menor o igual que el de la derecha.

Después de evaluar una condición, esta devuelve un valor lógico verdadero o falso. Esto nos permite trabajar con condiciones como si fueran valores y combinarlas para crear condiciones más complejas. JavaScript ofrece los siguientes operadores lógicos con este propósito.

- **!** (negación) permuta el estado de la condición. Si la condición es verdadera, devuelve falso, y viceversa.
- **&&** (y) comprueba dos condiciones y devuelve verdadero si ambas son verdaderas.
- **||** (o) comprueba dos condiciones y devuelve verdadero si una o ambas son verdaderas.

El operador lógico ! invierte el estado de la condición. Si la condición se evalúa como verdadera, el estado final será falso, y las instrucciones entre llaves no se ejecutarán.

```
var mivariable = 9;
if (!(mivariable < 10)) {
  alert("El número es menor que 10");
}
```

Listado 4-37: *Invirtiendo el resultado de la condición*

El código del Listado 4-37 no muestra ningún mensaje en la pantalla. El valor 9 es aún menor que 10, pero debido a que alteramos la condición con el operador !, el resultado final es falso, y la función alert() no se ejecuta.

Para que el operador trabaje sobre el estado de la condición y no sobre los valores que estamos comparando, debemos encerrar la condición entre paréntesis. Debido a los paréntesis, la condición se evalúa en primer lugar y luego el estado que devuelve se invierte con el operador! .

Los operadores && (y) y || (o) trabajan de un modo diferente. Estos operadores calculan el resultado final basándose en los resultados de las condiciones involucradas. El operador && (y) devuelve verdadero solo si las condiciones a ambos lados devuelven verdadero, y el operador || (o) devuelve verdadero si una o ambas condiciones devuelven verdadero. Por ejemplo, el siguiente código ejecuta la función alert() solo cuando la edad es menor de 21 y el valor de la variable inteligente es igual a "SI" (debido a que usamos el operador &&, ambas condiciones tienen que ser verdaderas para que la condición general sea verdadera).

```
var inteligente = "SI";
var edad = 19;
if (edad < 21 && inteligente == "SI") {
   alert("Juan está autorizado");
}
```

Listado 4-38: *Comprobando múltiples condiciones con operadores lógicos*

Si asumimos que nuestro ejemplo solo considera dos valores para la variable inteligente, "SI" y "NO", podemos convertirla en una variable booleana. Debido a que los valores booleanos son valores lógicos, no necesitamos compararlos con nada. El siguiente código simplifica el ejemplo anterior usando una variable booleana.

```
var inteligente = true;
var edad = 19;
if (edad < 21 && inteligente) {
   alert("Juan está autorizado");
}
```

Listado 4-39: *Usando valores booleanos como condiciones*

JavaScript es bastante flexible en cuanto a los valores que podemos usar para establecer condiciones. El lenguaje es capaz de determinar una condición basándose en los valores de cualquier variable. Por ejemplo, una variable con un número entero devolverá falso si el valor es 0 o verdadero si el valor es diferente de 0.

```
var edad = 0;
if (edad) {
   alert("Juan está autorizado");
}
```

Listado 4-40: *Usando números enteros como condiciones*

El código de la instrucción if del Listado 4-40 no se ejecuta porque el valor de la variable edad es 0 y, por lo tanto, el estado de la condición se considera falso. Si almacenamos un valor diferente dentro de esta variable, la condición será verdadera y el mensaje se mostrará en la pantalla.

Las variables con cadenas de caracteres vacías también devuelven falso. El siguiente ejemplo comprueba si se ha asignado una cadena de caracteres a una variable y muestra su valor solo si la cadena no está vacía.

```
var nombre = "Juan";
if (nombre) {
   alert(nombre + " está autorizado");
}
```

Listado 4-41: *Usando cadenas de caracteres como condiciones*

Ejercicio 8:reemplace el código en su archivo micodigo.js con el código del Listado 4-41 y abra el documento del Listado 4-5 en su navegador.La ventana emergente muestra el mensaje "Juan está autorizado". Asigne una cadena vacía a la variable **`nombre`**. La instrucción **`if`** ahora considera falsa la condición y no se muestra ningún mensaje en pantalla.

A veces debemos ejecutar instrucciones para cada estado de la condición (verdadero o falso). JavaScript incluye la instrucción if else para ayudarnos en estas situaciones. Las instrucciones se presentan en dos bloques de código delimitados por llaves. El bloque precedido por if se ejecuta cuando la condición es verdadera y el bloque precedido por else se ejecuta en caso contrario.

```
var mivariable = 21;
if (mivariable < 10) {
  alert("El número es menor que 10");
} else {
  alert("El numero es igual o mayor que 10");
}
```

Listado 4-42: *Comprobando dos condiciones con* `if else`

En este ejemplo, el código considera dos condiciones: cuando el número es menor que 10 y cuando el número es igual o mayor que 10. Si lo que necesitamos es comprobar múltiples condiciones, en lugar de las instrucciones if else podemos usar la instrucción switch. Esta instrucción evalúa una expresión (generalmente una variable), compara el resultado con múltiples valores y ejecuta las instrucciones correspondientes al valor que coincide con la expresión. La sintaxis incluye la palabra clave switch seguida de la expresión entre paréntesis. Los posibles valores se listan usando la palabra clave case, tal como muestra el siguiente ejemplo.

```
var mivariable = 8;
switch(mivariable) {
  case 5:
    alert("El número es cinco");
    break;
  case 8:
    alert("El número es ocho");
    break;
  case 10:
    alert("El número es diez");
    break;
  default:
    alert("El número es " + mivariable);
}
```

Listado 4-43: *Comprobando un valor con la instrucción* `switch`

En el ejemplo del Listado 4-43, la instrucción switch evalúa la variable mivariable y luego compara su valor con el valor de cada caso. Si el valor es 5, por ejemplo, el control se transfiere al primer case, y la función alert() muestra el texto "El número es cinco" en la pantalla. Si el primer **`case`** no coincide con el valor de la variable, se evalúa el siguiente caso, y así sucesivamente. Si ningún caso coincide con el valor, se ejecutan las instrucciones en el caso **`default`**.

En JavaScript, una vez que se encuentra una coincidencia, las instrucciones en ese caso se ejecutan junto con las instrucciones de los casos siguientes. Este es el comportamiento por defecto, pero normalmente no lo que nuestro código necesita. Por esta razón, JavaScript incluye la instrucción break. Para evitar que el sistema ejecute las instrucciones de cada caso después de que se encuentra una coincidencia, tenemos que incluir la instrucción break al final de cada caso.

Las instrucciones switch e if son útiles pero realizan una tarea sencilla: evalúan una expresión, ejecutan un bloque de instrucciones de acuerdo al resultado y al final devuelven el control al código principal. En ciertas situaciones esto no es suficiente. A veces tenemos que ejecutar las instrucciones varias veces para la misma condición o evaluar la condición nuevamente cada vez que se termina un proceso. Para estas situaciones, contamos con dos
instrucciones: for y while.

La instrucción for ejecuta el código entre llaves mientras la condición es verdadera. Usa la sintaxis for(inicialización; condición; incremento). El primer parámetro establece los valores iniciales del bucle, el segundo parámetro es la condición que queremos comprobar y el último parámetro es una instrucción que determina cómo van a evolucionar los valores iniciales en cada ciclo.

```
var total = 0;
for (var f = 0; f < 5; f++) {
  total += 10;
}
alert("El total es: " + total);  // "El total es: 50"
```

Listado 4-44: *Creando un bucle con la instrucción* `for`

En el código del Listado 4-44 declaramos una variable llamada f para controlar el bucle y asignamos el número 0 como su valor inicial. La condición en este ejemplo comprueba si el valor de la variable f es menor que 5. En caso de ser verdadera, se ejecuta el código entre llaves. Después de esto, el intérprete ejecuta el último parámetro de la instrucción for, el cual suma 1 al valor actual de f (f++), y luego comprueba la condición nuevamente (en cada ciclo f se incrementa en 1). Si la condición es aún verdadera, las instrucciones se ejecutan una vez más. Este proceso continúa hasta que f alcanza el valor 5, lo cual vuelve falsa la condición (5 no es menor que 5) y el bucle se interrumpe.

Dentro del bucle for del Listado 4-44, sumamos el valor 10 al valor actual de la variable total. Los bucles se usan frecuentemente de esta manera para hacer evolucionar el valor de una variable de acuerdo a resultados anteriores. Por ejemplo, podemos usar el bucle for para sumar todos los valores de un array.

```
var total = 0;
var lista = [23, 109, 2, 9];
for (var f = 0; f < 4; f++) {
  total += lista[f];
}
alert("El total es: " + total);  // "El total es: 143"
```

Listado 4-45: *Iterando sobre los valores de un array*

Para leer todos los valores de un array, tenemos que crear un bucle que va desde el índice del valor inicial a un valor que coincide con el índice del último valor del array. En este caso, el array lista contiene cuatro elementos y, por lo tanto, los índices correspondientes van de 0 a 3. El bucle lee el valor en el índice 0, lo suma al valor actual de la variable total y luego avanza hacia el siguiente valor en el array hasta que el valor de f es igual a 4 (no existe un valor en el índice 4). Al final, todos los valores del array se suman a la variable total y el resultado se muestra en pantalla.

Lo básico: en el ejemplo del Listado 4-45, hemos podido configurar el bucle porque conocemos el número de valores dentro del array, pero esto no es siempre posible. A veces no contamos con esta información durante el desarrollo de la aplicación, ya sea porque el array se crea cuando la página se carga o porque los valores los introduce el usuario. Para trabajar con arrays dinámicos, JavaScript ofrece la propiedad **`length`**. Esta propiedad devuelve la cantidad de valores dentro de un array. Estudiaremos esta propiedad y los objetos **`Array`** más adelante en este capítulo.

La instrucción for es útil cuando podemos determinar ciertos requisitos, como el valor inicial del bucle o el modo en que evolucionarán esos valores en cada ciclo. Cuando esta información es poco clara, podemos utilizar la instrucción while. La instrucción while solo requiere la declaración de la condición entre paréntesis y el código a ser ejecutado entre llaves. El bucle se ejecuta constantemente hasta que la condición es falsa.

```
var contador = 0;
while(contador < 100) {
  contador++;
}
alert("El valor es: " + contador);  // "El valor es: 100"
```

Listado 4-46: *Usando la instrucción* `while`

El ejemplo del Listado 4-46 es sencillo. La instrucción entre llaves se ejecuta mientras el valor de la variable contador es menor que 100. Esto significa que el bucle se ejecutará 100 veces (cuando el valor de contador es 99, la instrucción se ejecuta una vez más y, por lo tanto, el valor final de la variable es 100).

Si la primera vez que la condición se evalúa devuelve un valor falso (por ejemplo, cuando el valor inicial de contador ya es mayor de 99), el código entre llaves nunca se ejecuta. Si queremos que las instrucciones se ejecuten al menos una vez, sin importar cuál sea el resultado de la condición, podemos usar una implementación diferente del bucle while llamada do while. La instrucción do while ejecuta las instrucciones entre llaves y luego comprueba la condición, lo cual garantiza que las instrucciones se ejecutarán al menos una vez. La sintaxis es similar, solo tenemos que preceder las llaves con la palabra clave do y declarar la palabra clave while con la condición al final.

```
var contador = 150;
do {
  contador++;
} while(contador < 100);
alert("El valor es: " + contador);  // "El valor es: 151"
```

Listado 4-47: *Usando la instrucción* `do while`

En el ejemplo del Listado 4-47, el valor inicial de la variable contador es mayor de 99, pero debido a que usamos el bucle do while, la instrucción entre llaves se ejecuta una vez y, por lo tanto, el valor final de contador es 151 (150 + 1 = 151).

1.8. Instrucciones de transferencia de control

Los bucles a veces se deben interrumpir. JavaScript ofrece múltiples instrucciones para detener la ejecución de bucles y condicionales. Las siguientes son las que más se usan.

continue—Esta instrucción interrumpe el ciclo actual y avanza hacia el siguiente. El sistema ignora el resto de instrucciones del bucle después de que se ejecuta esta instrucción.

break—Esta instrucción interrumpe el bucle. Todas las instrucciones restantes y los ciclos pendientes se ignoran después de que se ejecuta esta instrucción.

La instrucción continue se aplica cuando no queremos ejecutar el resto de las instrucciones entre llaves, pero queremos seguir ejecutando el bucle.

```
var lista = [2, 4, 6, 8];

var total = 0;
for (var f = 0; f < 4; f++) {
  var numero = lista[f];
  if (numero == 6) {
    continue;
  }
  total += numero;
}
alert("El total es: " + total);  // "El total es: 14"
```

Listado 4-48: *Saltando hacia el siguiente ciclo del bucle*

La instrucción if dentro del bucle for del Listado 4-48 compara el valor de numero con el valor 6. Si el valor del array que devuelve la primera instrucción del bucle es 6, se ejecuta la instrucción continue, la última instrucción del bucle se ignora, y el bucle avanza hacia el siguiente valor en el array lista. En consecuencia, todos los valores del array se suman a la variable total excepto el número 6.

A diferencia de continue, la instrucción break interrumpe el bucle completamente, delegando el control a la instrucción declarada después de bucle.

```
var lista = [2, 4, 6, 8];

var total = 0;
for (var f = 0; f < 4; f++) {
  var numero = lista[f];
  if (numero == 6) {
    break;
  }
  total += numero;
}
alert("El total es: " + total);  // "El total es: 6"
```

Listado 4-49: *Interrumpiendo el bucle*

Nuevamente, la instrucción if del Listado 4-49 compara el valor de numero con el valor 6, pero esta vez ejecuta la instrucción break cuando los valores coinciden. Si el número del array que devuelve la primera instrucción es 6, la instrucción break se ejecuta y el bucle termina, sin importar cuántos valores quedaban por leer en el array. En consecuencia, solo los valores ubicados antes del número 6 se suman al valor de la variable total.

Capítulo 2
Funciones

2.1. Introducción a las funciones

Las funciones son bloques de código identificados con un nombre. La diferencia entre las funciones y los bloques de código usados en los bucles y los condicionales estudiados anteriormente es que no hay que satisfacer ninguna condición; las instrucciones dentro de una función se ejecutan cada vez que se llama a la función. Las funciones se llaman (ejecutadas) escribiendo el nombre seguido de paréntesis. Esta llamada se puede realizar desde cualquier parte del código y cada vez que sea necesario, lo cual rompe completamente el procesamiento secuencial del programa. Una vez que una función es llamada, la ejecución del programa continúa con las instrucciones dentro de la función (sin importar dónde se localiza en el código) y solo devuelve a la sección del código que ha llamado la función cuando la ejecución de la misma ha finalizado.

2.2. Declarando funciones

Las funciones se declaran usando la palabra clave function, el nombre seguido de paréntesis, y el código entre llaves. Para llamar a la función (ejecutarla), tenemos que declarar su nombre con un par de paréntesis al final, como mostramos a continuación.

```
function mostrarMensaje() {
  alert("Soy una función");
}
mostrarMensaje();
```

Listado 4-50: *Declarando funciones*

Las funciones se deben primero declarar y luego ejecutar. El código del Listado 4-50 declara una función llamada mostrarMensaje() y luego la llama una vez. Al igual que con las variables, el intérprete de JavaScript lee la función, almacena su contenido en memoria, y asigna una referencia al nombre de la función. Cuando llamamos a la función por su nombre, el intérprete comprueba la referencia y lee la función en memoria. Esto nos permite llamar a la función todas las veces que sea necesario, como los hacemos en el siguiente ejemplo.

```
var total = 5;
function calcularValores(){
  total = total * 2;
}
for(var f = 0; f < 10; f++){
  calcularValores();
}
alert("El total es: " + total);  // "El total es: 5120"
```

Listado 4-51: *Procesando datos con funciones*

El ejemplo del Listado 4-51 combina diferentes instrucciones ya estudiadas. Primero declara una variable y le asigna el valor 5. Luego, se declara una función llamada calcularValores() (pero no se ejecuta). A continuación, una instrucción for se usa para crear un bucle que será ejecutado mientras el valor de la variable f sea menor que 10. La instrucción dentro del bucle llama a la función calcularValores(), por lo que la función se ejecuta en cada ciclo. Cada vez que la función se ejecuta, el valor actual de total se multiplica por 2, duplicando su valor en cada ocasión.

2.3. Ámbito

En JavaScript, las instrucciones que se encuentran fuera de una función se considera que están en el ámbito global. Este es el espacio en el que escribimos las instrucciones hasta que se define una función u otra clase de estructura de datos. Las variables definidas en el ámbito global tienen un alcance global y, por lo tanto, se pueden usar desde cualquier parte del código, pero las declaradas dentro de las funciones tienen un alcance local, lo que significa que solo se pueden usar dentro de la función en la que se han declarado. Esta es otra ventaja de las funciones; son lugares especiales en el código donde podemos almacenar información a la que no se podrá acceder desde otras partes del código. Esta segregación nos ayuda a evitar generar duplicados que pueden conducir a errores, como sobrescribir el valor de una variable cuando el valor anterior aún era requerido por la aplicación.

El siguiente ejemplo ilustra cómo se definen los diferentes ámbitos y qué debemos esperar cuando accedemos desde un ámbito a variables que se han definido en un ámbito diferente.

```
var variableGlobal = 5;
function mifuncion(){
  var variableLocal = "El valor es ";
  alert(variableLocal + variableGlobal);  // "El valor es 5"
}
mifuncion();
alert(variableLocal);
```

Listado 4-52: *Declarando variables globales y locales*

El código del Listado 4-52 declara una función llamada mifuncion() y dos variables, una en el ámbito global llamada variableGlobal y otra dentro de la función llamada variableLocal. Cuando se ejecuta la función, el código concatena las variables variableLocal y variableGlobal, y muestra la cadena de caracteres obtenida en la pantalla. Debido a que variableGlobal es una variable global, es accesible dentro de la función y, por lo tanto, su valor se agrega a la cadena de caracteres, pero cuando intentamos mostrar el valor de variableLocal fuera de la función, nos devuelve un error (la ventana emergente no se muestra). Esto se debe a que variableLocal se ha definido dentro de la función y, por lo tanto, no es accesible desde el ámbito global.

Lo básico: los navegadores informan de los errores producidos por el código JavaScript en una consola oculta. Si necesita ver los errores porque genera su código, tiene que abrir esta consola desde las opciones del menú del navegador (Herramientas, en Google Chrome). Al final de este capítulo estudiaremos más estas consolas y cómo controlar errores.

Debido a que las variables declaradas en diferentes ámbitos se consideran diferentes variables, dos variables con el mismo nombre, una en el ámbito global y otra en el ámbito local (dentro de una función), se considerarán dos variables distintas (se les asigna un espacio de memoria diferente).

```
var mivariable = 5;
function mifuncion(){
  var mivariable = "Esta es una variable local";
  alert(mivariable);
}
mifuncion();
alert(mivariable);
```

Listado 4-53: *Declarando dos variables con el mismo nombre*

En el código del Listado 4-53, declaramos dos variables llamadas mivariable, una en el ámbito global y la otra dentro de la función mifuncion(). También incluimos dos funciones alert() en el código, una dentro de la función mifuncion() y otra en el ámbito global al final del código. Ambas muestran el contenido de la variable mivariable, pero referencian distintas variables. La variable dentro de la función está referenciando un espacio en memoria que contiene la cadena de caracteres "Esta es una variable local", mientras que la variable en el ámbito global está referenciando un espacio en memoria que contiene el valor 5.

Lo básico: las variables globales también se pueden crear desde las funciones. Omitir la palabra clave **`var`** cuando declaramos una variable dentro de una función es suficiente para configurar esa variable como global.

Las variables globales son útiles cuando varias funciones deben compartir valores, pero debido a que son accesibles desde cualquier parte del código, siempre existe la posibilidad de sobrescribir sus valores por accidente desde otras instrucciones, o incluso desde otros códigos (todos los códigos JavaScript incluidos en el documento comparten el mismo ámbito global).

Por consiguiente, usar variables globales desde una función no es una buena idea. Una mejor alternativa es enviar valores a las funciones cuando son llamadas.

Para poder recibir un valor, la función debe incluir un nombre entre los paréntesis con el que representar el valor. Estos nombres se denominan parámetros. Cuando la función se ejecuta, estos parámetros se convierten en variables que podemos leer desde dentro de la función y así acceder a los valores recibidos.

```
function mifuncion(valor) {
  alert(valor);
}
mifuncion(5);
```

Listado 4-54: *Enviando un valor a una función*

En el ejemplo del Listado 4-54, dejamos de usar variables globales. El valor a procesar se envía a la función cuando es llamada y esta lo recibe a través de su parámetro. Cuando se llama a la función, el valor entre los paréntesis de la llamada (5) se asigna a la variable value creada para recibirlo, y esta variable se lee dentro de la función para mostrar el valor en pantalla.

Lo básico: los nombres declarados entre los paréntesis de la función para recibir valores se llaman *parámetros*. Por otro lado, los valores especificados en la llamada se denominan *atributos*. En estos términos, podemos decir que la llamada a la función tiene atributos que se envían a la función y se reciben mediante sus parámetros.

La ventaja de usar funciones es que podemos ejecutar sus instrucciones una y otra vez, y como podemos enviar diferentes valores en cada llamada, el resultado obtenido en cada una de ellas será diferente. El siguiente ejemplo llama a la función mifuncion() dos veces, pero en cada oportunidad envía un valor diferente para ser procesado.

```
function mifuncion(valor) {
  alert(valor);
}
mifuncion(5);
mifuncion(25);
```

Listado 4-55: *Llamando a la misma función con diferentes valores*

El intérprete ejecuta la primera llamada con el valor 5 y cuando la ejecución de la función finaliza, se llama nuevamente con el valor 25. En consecuencia, se abren dos ventanas emergentes, una con el valor 5 y la otra con el valor 25.

En este y los ejemplos anteriores, enviamos números enteros a la función, pero también podemos enviar el valor actual de una variable.

```
var contador = 100;
function mifuncion(valor) {
  alert(valor);
}
mifuncion(contador);
```

Listado 4-56: *Enviando el valor de una variable a una función*

Esta vez incluimos la variable contador en la llamada en lugar de un número. El intérprete lee esta variable y envía su valor a la función. El resto del proceso es el mismo: la función recibe el valor, lo asigna a la variable valor y lo muestra en pantalla.

Las funciones también pueden recibir múltiples valores. Todo lo que tenemos que hacer es declarar los valores y parámetros separados por comas.

```
var contador = 100;
var items = 5;

function mifuncion(valor1, valor2) {
  var total = valor1 + valor2;
  alert(total);  // 105
}
mifuncion(contador, items);
```

Listado 4-57: *Enviando múltiples valores a la función*

En el ejemplo del Listado 4-57, sumamos los valores recibidos por la función y mostramos el resultado en pantalla, pero a veces este resultado se requiere fuera de la función. Para enviar valores desde la función al ámbito global, JavaScript incluye la instrucción return. Esta instrucción determina el valor a devolver al código que ha llamado a la función.

Si queremos procesar el valor que devuelve la función, tenemos que asignar dicha función a una variable. El intérprete primero ejecuta la función y luego asigna el valor que devuelve la función a la variable, tal como muestra el siguiente ejemplo.

```
var contador = 100;
var items = 5;
function mifuncion(valor1, valor2) {
  var total = valor1 + valor2;
  return total;
}
var resultado = mifuncion(contador, items);
alert(resultado);
```

Listado 4-58: *Devolviendo valores desde funciones*

El código del Listado 4-58 define la misma función mifuncion() usada anteriormente, pero esta vez el valor producido por la función no se muestra en la pantalla, sino que se devuelve con la instrucción return. De regreso al ámbito global, el valor devuelto por la función se asigna a la variable resultado y el contenido de esta variable se muestra en pantalla.

La variable miresultado se ha declarado al comienzo del código, pero no le hemos asignamos ningún valor. Esto es aceptable, e incluso recomendado. Es una buena práctica declarar todas las variables con las que vamos a trabajar al comienzo para evitar confusión y poder identificar cada una de ellas más adelante desde otras partes del código.

La instrucción return finaliza la ejecución de la función. Cualquier instrucción declarada después de que se devuelve un valor no se ejecutará. Por esta razón, la instrucción return normalmente se declara al final de la función, pero esto no es obligatorio. Podemos devolver un valor desde cualquier parte del código si tenemos condiciones que satisfacer. Por ejemplo, la siguiente función devuelve el resultado de la suma de dos valores si el total es mayor que 100, o devuelve el valor 0 en caso contrario.

```
var contador = 100;
var items = 5;
function mifuncion(valor1, valor2) {
  var total = valor1 + valor2;
  if (total > 100) {
    return total;
  } else {
    return 0;
  }
}
var resultado = mifuncion(contador, items);
alert(resultado);
```

Listado 4-59: *Devolviendo diferentes valores desde una función*

2.4. Funciones anónimas

Otra manera de declarar una función es usando funciones anónimas. Las funciones anónimas son funciones sin un nombre o identificador. Debido a esto, se pueden pasar a otras funciones o asignar a variables. Cuando una función anónima se asigna a una variable, el nombre de la variable es el que usamos para llamar a la función, tal como hacemos en el siguiente ejemplo.

```
var mifuncion = function(valor) {
  valor = valor * 2;
  return valor;
};
var total = 2;
for (var f = 0; f < 10; f++) {
  total = mifuncion(total);
}
alert("El total es " + total);  // "El total es 2048"
```

Listado 4-60: *Declarando funciones anónimas*

En el ejemplo del Listado 4-60, declaramos una función anónima que recibe un valor, lo multiplica por 2 y devuelve el resultado. Debido a que la función se asigna a una variable, podemos usar el nombre de la variable para llamarla, por lo que después de que se define la función, creamos un bucle for que llama a la función mifuncion() varias veces con el valor actual de la variable total. La instrucción del bucle asigna el valor que devuelve la función de vuelta a la variable total, duplicando su valor en cada ciclo.

Las funciones anónimas se pueden ejecutar al instante agregando paréntesis al final de su declaración. Esto es útil cuando queremos asignar el resultado de una operación compleja a una variable. La función procesa la operación y devuelve el resultado. En este caso, no es la función lo que se asigna a la variable, sino el valor que devuelve la misma.

```
var mivalor = function(valor) {
  valor = valor * 2;
  return valor;
}(35);
alert("El valor es " + mivalor);  // "El valor es 70"
```

Listado 4-61: *Ejecutando funciones anónimas*

La función del Listado 4-61 se define y ejecuta tan pronto como el intérprete procesa la instrucción. La función recibe el valor 35, lo multiplica por 2 y devuelve el resultado, que se asigna a la variable mivalor.

Lo básico: las funciones anónimas son extremadamente útiles en JavaScript porque nos permiten definir complicados patrones de programación, necesarios para construir aplicaciones profesionales. Estudiaremos ejemplos prácticos del uso de estos tipos de funciones y algunos patrones disponibles en JavaScript en próximos capítulos.

2.5. Funciones estándar

Además de las funciones que podemos crear nosotros mismos, también tenemos acceso a funciones predefinidas por JavaScript. Estas funciones realizan procesos que simplifican tareas complejas. Las siguientes son las que más se usan.

isNaN(valor**)**—Esta función devuelve **`true`** (verdadero) si el valor entre paréntesis no es un número.

parseInt(valor**)**—Esta función convierte una cadena de caracteres con un número en un número entero que podemos procesar en operaciones aritméticas.

parseFloat(valor**)**—Esta función convierte una cadena de caracteres con un número en un número decimal que podemos procesar en operaciones aritméticas.

encodeURIComponent(valor**)**—Esta función codifica una cadena de caracteres. Se utiliza para codificar los caracteres de un texto que puede crear problemas cuando se inserta en una URL.

decodeURIComponent(valor**)**—Esta función decodifica una cadena de caracteres.

Las funciones estándar son funciones globales que podemos llamar desde cualquier parte del código; solo tenemos que llamarlas como lo hacemos con cualquier otra función con los valores que queremos procesar entre paréntesis.

```
var mivalor = "Hola";
if (isNaN(mivalor)) {
  alert(mivalor + " no es un número");
} else {
  alert(mivalor + " es un número");
}
```

Listado 4-62: *Comprobando si un valor es un número o no*

La función isNaN() devuelve un valor booleano, por lo que podemos usarla para establecer una condición. El intérprete primero llama a la función y luego ejecuta los bloques de código definidos por las instrucciones if else dependiendo del resultado. En este caso, el valor de la variable es una cadena de caracteres, por lo que la función isNaN() devuelve el valor true y el mensaje "Hola no es un número" se muestra en pantalla.

La función isNaN() devuelve el valor false no solo cuando la variable contiene un número, sino además cuando contiene una cadena de caracteres con un número. Esto significa que no podemos usar el valor en una operación aritmética porque podría ser una cadena de caracteres y el proceso no produciría el resultado esperado. Para asegurarnos de que el valor se puede incluir en una operación, tenemos que convertirlo en un valor numérico. Para este propósito, JavaScript ofrece dos funciones: parseInt() para números enteros y parseFloat() para números decimales.

```
var mivalor = "32";
if (isNaN(mivalor)) {
  alert(mivalor + " no es un número");
} else {
  var numero = parseInt(mivalor);
  numero = numero * 10;
  alert("El número es: " + numero);  // "El número es 320"
}
```

Listado 4-63: *Convirtiendo una cadena de caracteres en un número*

El código del Listado 4-63 comprueba si el valor de la variable mivalor es un número o no, como hemos hecho antes, pero esta vez el valor se convierte en un valor numérico si se encuentra un número. Después de que el valor se extrae de la cadena de caracteres, lo usamos para realizar una multiplicación y mostrar el resultado en pantalla.

Otra función estándar útil es encodeURIComponent(). Con esta función podemos preparar una cadena de caracteres para ser incluida en una URL. El problema con las URL es que le otorgan un significado especial a algunos caracteres, como ? o &, como hemos visto en capítulos anteriores (ver Figura 2-43). Debido a que los usuarios no conocen estas restricciones, tenemos que codificar las cadenas de caracteres antes de incluirlas en una URL cada vez que las introduce el usuario o provienen de una fuente que no es fiable.

```
var nombre = "Juan Perez";
var codificado = encodeURIComponent(nombre);
var miURL = "http://www.ejemplo.com/contacto.html?nombre=" +
codificado;
alert(miURL);
```

Listado 4-64: *Codificando una cadena de caracteres para incluirla en una URL*

El código del Listado 4-64 agrega el valor de la variable nombre a una URL. En este ejemplo, asumimos que el valor de la variable lo ha definido el usuario y, por lo tanto, lo codificamos con la función encodeURIComponent() para asegurarnos de que la URL final es válida. La función analiza la cadena de caracteres y reemplaza cada carácter conflictivo por un número hexadecimal precedido por el carácter %. En este caso, el único carácter que requiere codificación es el espacio entre el nombre y el apellido. La URL resultante es **http://www.ejemplo.com/contacto.html?nombre=Juan%20Perez.**

Capítulo 3
Objetos

3.1. Objetos

Los objetos son estructuras de información capaces de contener variables (llamadas propiedades), así como funciones (llamadas métodos). Debido a que los objetos almacenan valores junto con funciones, son como programas independientes que se comunican entre sí para realizar tareas comunes.

La idea detrás de los objetos en programación es la de simular el rol de los objetos en la vida real. Un objeto real tiene propiedades y realiza acciones. Por ejemplo, una persona tiene un nombre y una dirección postal, pero también puede caminar y hablar. Las características y la funcionalidad son parte de la persona y es la persona la que define cómo va a caminar y lo que va a decir. Organizando nuestro código de esta manera, podemos crear unidades de procesamiento independientes capaces de realizar tareas y que cuentan con toda la información que necesitan para hacerlo. Por ejemplo, podemos crear un objeto que controla un botón, muestra su título, y realiza una tarea cuando se pulsa el botón. Debido a que toda la información necesaria para presentar y controlar el botón se almacena dentro del objeto, el resto del código no necesita saber cómo hacerlo. Siempre y cuando conozcamos los métodos provistos por el objeto y los valores devueltos, el código dentro del objeto se puede actualizar o reemplazar por completo sin afectar el resto del programa.

Poder crear unidades de procesamiento independientes, duplicar esas unidades tantas veces como sea necesario, y modificar sus valores para adaptarlos a las circunstancias actuales, son las principales ventajas introducidas por los objetos y la razón por la que la programación orientada a objetos es el paradigma de programación disponible más popular. JavaScript se creó en torno al concepto de objetos y, por lo tanto, entender objetos es necesario para entender el lenguaje y sus posibilidades.

3.2. Declarando objetos

Existen diferentes maneras de declarar objetos en JavaScript, pero la más sencilla es usar notación literal. El objeto se declara como cualquier otra variable usando la palabra clave var, y las propiedades y métodos que definen el objeto se declaran entre llaves usando dos puntos después del nombre y una coma para separar cada declaración.

```
var miobjeto = {
  nombre: "Juan",
  edad: 30
};
```

Listado 4-65: *Creando objetos*

En el ejemplo del Listado 4-65 declaramos el objeto miobjeto con dos propiedades: nombre y edad. El valor de la propiedad nombre es "Juan" y el valor de la propiedad edad es 30.

A diferencia de las variables, no podemos acceder a los valores de las propiedades de un objeto usando solo sus nombres; también tenemos que especificar el nombre del objeto al que pertenecen usando notación de puntos o corchetes.

```
var miobjeto = {
  nombre: "Juan",
  edad: 30
};
var mensaje = "Mi nombre es " + miobjeto.nombre + "\r\n";
mensaje += "Tengo " + miobjeto["edad"] + " años";
alert(mensaje);
```

Listado 4-66: *Accediendo propiedades*

En el Listado 4-66, implementamos ambas técnicas para acceder a los valores de las propiedades del objeto y crear el mensaje que vamos a mostrar en la pantalla. El uso de cualquiera de estas técnicas es irrelevante, excepto en algunas circunstancias. Por ejemplo, cuando necesitamos acceder a la propiedad a través del valor de una variable, tenemos que usar corchetes.

```
var nombrePropiedad = "nombre";
var miobjeto = {
  nombre: "Juan",
  edad: 30
};
alert(miobjeto[nombrePropiedad]);   // "Juan"
```

Listado 4-67: *Accediendo propiedades usando variables*

En el Listado 4-67, no podríamos haber accedido a la propiedad usando notación de puntos (miobjeto.nombrePropiedad) porque el intérprete habría intentado acceder a una propiedad llamada nombrePropiedad que no existe. Usando corchetes, primero la variable se resuelve y luego se accede al objeto con su valor ("nombre") en lugar de su nombre.

También es necesario acceder a una propiedad usando corchetes cuando su nombre se considera no válido para una variable (incluye caracteres no válidos, como un espacio, o comienza con un número). En el siguiente ejemplo, el objeto incluye una propiedad cuyo nombre se ha declarado con una cadena de caracteres. Está permitido declarar nombres de propiedades con cadena de caracteres, pero como este nombre contiene un espacio, el código miobjeto.mi edad produciría un error, por lo que tenemos que usar corchetes para acceder a esta propiedad.

```
var mivariable = "nombre";
var miobjeto = {
  nombre: "Juan",
  'mi edad': 30
};
alert(miobjeto['mi edad']);  // 30
```

Listado 4-68: *Accediendo propiedades con nombres no válidos*

Además de leer los valores de las propiedades, también podemos asignar nuevas propiedades al objeto o modificarlas usando notación de puntos. En el siguiente ejemplo, modificamos el valor de la propiedad nombre y agregamos una nueva propiedad llamada trabajo.

```
var miobjeto = {
  nombre: "Juan",
  edad: 30
};
miobjeto.nombre = "Martín";
miobjeto.trabajo = "Programador";
alert(miobjeto.nombre + " " + miobjeto.edad + " " + miobjeto.trabajo);
```

Listado 4-69: *Actualizando valores y agregando nuevas propiedades a un objeto*

Los objetos también pueden contener otros objetos. En el siguiente ejemplo, asignamos un objeto a la propiedad de otro objeto.

```
var miobjeto = {
  nombre: "Juan",
  edad: 30,
  motocicleta: {
    modelo: "Susuki",
    fecha: 1981
  }
};
alert(miobjeto.nombre + " tiene una " + miobjeto.motocicleta.modelo);
```

Listado 4-70: *Creando objetos dentro de objetos*

El objeto miobjeto en el código del Listado 4-70 incluye una propiedad llamada motocicleta cuyo valor es otro objeto con las propiedades modelo y fecha. Si queremos acceder a estas propiedades, tenemos que indicar el nombre del objeto al que pertenecen (motocicleta) y el nombre del objeto al que ese objeto pertenece (miobjeto). Los nombres se concatenan con notación de puntos en el orden en el que se han incluido en la jerarquía. Por ejemplo, la propiedad modelo está dentro de la propiedad motocicleta que a la vez está dentro de la propiedad miobjeto. Por lo tanto, si queremos leer el valor de la propiedad modelo, tenemos que escribir miobjeto.motocicleta.modelo.

3.3. Métodos

Como mencionamos anteriormente, los objetos también pueden incluir funciones. Las funciones dentro de los objetos se llaman métodos. Los métodos tienen la misma sintaxis que las propiedades: requieren dos puntos después del nombre y una coma para separar cada declaración, pero en lugar de valores, debemos asignarles funciones anónimas.

```
var miobjeto = {
  nombre: "Juan",
  edad: 30,
  mostrardatos: function() {
    var mensaje = "Nombre: " + miobjeto.nombre + "\r\n";
    mensaje += "Edad: " + miobjeto.edad;
    alert(mensaje);
  },
  cambiarnombre: function(nombrenuevo) {
    miobjeto.nombre = nombrenuevo;
  }
};
miobjeto.mostrardatos();  // "Nombre: Juan Edad: 30"
miobjeto.cambiarnombre("José");
miobjeto.mostrardatos();  // "Nombre: José Edad: 30"
```

Listado 4-71: *Declarando y ejecutando métodos*

En este ejemplo, agregamos dos métodos al objeto: mostrardatos() y cambiarnombre(). El método mostrardatos() muestra una ventana emergente con los valores de las propiedades nombre y edad, y el método cambiarnombre() asigna el valor recibido por su parámetro a la propiedad nombre. Estos son dos métodos independientes que trabajan sobre las mismas propiedades, uno lee sus valores y el otro les asigna nuevos. Para ejecutar los métodos, usamos notación de puntos y paréntesis después del nombre, como hacemos con funciones.

Al igual que las funciones, los métodos también pueden devolver valores. En el siguiente ejemplo, modificamos el método cambiarnombre() para devolver el nombre anterior después de que se reemplaza por el nuevo.

```
var miobjeto = {
  nombre: "Juan",
  edad: 30,
  mostrardatos: function() {
    var mensaje = "Nombre: " + miobjeto.nombre + "\r\n";
    mensaje += "Edad: " + miobjeto.edad;
    alert(mensaje);
  },
  cambiarnombre: function(nombrenuevo) {
    var nombreviejo = miobjeto.nombre;
    miobjeto.nombre = nombrenuevo;
    return nombreviejo;
  }
};
var anterior  =  miobjeto.cambiarnombre("José");
alert("El nombre anterior era: " + anterior);  // "Juan"
```

Listado 4-72: *Devolviendo valores desde métodos*

El nuevo método cambiarnombre() almacena el valor actual de la propiedad nombre en una variable temporal llamada nombreviejo para poder devolver el valor anterior después de que el nuevo se asigna a la propiedad.

3.4. La palabra clave this

En los últimos ejemplos, mencionamos el nombre del objeto cada vez que queríamos modificar sus propiedades desde los métodos. Aunque esta técnica funciona, no es una práctica recomendada. Debido a que el nombre del objeto queda determinado por el nombre de la variable al que se asigna el objeto, el mismo se puede modificar sin advertirlo. Además, como veremos más adelante, JavaScript nos permite crear múltiples objetos desde la misma definición o crear nuevos objetos a partir de otros, lo cual produce diferentes objetos que comparten la misma definición. Para asegurarnos de que siempre referenciamos al objeto con el que estamos trabajando, JavaScript incluye la palabra clave this. Esta palabra clave se usa en lugar del nombre del objeto para referenciar el objeto al que la instrucción pertenece. El siguiente ejemplo reproduce el código anterior, pero esta vez usamos la palabra clave this en lugar del nombre del objeto para referenciar sus propiedades. El resultado es el mismo que antes.

```
var miobjeto = {
  nombre: "Juan",
  edad: 30,
  mostrardatos: function() {
    var mensaje = "Nombre: " + this.nombre + "\r\n";
    mensaje += "Edad: " + this.edad;
    alert(mensaje);
  },
  cambiarnombre: function(nombrenuevo) {
    var nombreviejo = this.nombre;
    this.nombre = nombrenuevo;
    return nombreviejo;
  }
};
var anterior = miobjeto.cambiarnombre("José");
alert("El nombre anterior era: " + anterior);   // "Juan"
```

Listado 4-73: *Referenciando las propiedades del objeto con la palabra clave* `this`

IMPORTANTE: cada vez que queremos acceder a propiedades y métodos desde el interior de un objeto, debemos usar la palabra clave **`this`** para referenciar el objeto, pero si intentamos hacer lo mismo desde fuera del objeto, en su lugar estaremos referenciando el objeto global de JavaScript. La palabra clave **`this`** referencia el objeto en el que la instrucción se está ejecutando. Esta es la razón por la que en el código del Listado 4-73 solo usamos la palabra clave **`this`** dentro de los métodos del objeto **`miobjeto`**, pero las instrucciones en el ámbito global siguen usando el nombre del objeto.

3.5. Constructores

Usando notación literal podemos crear objetos individuales, pero si queremos crear copias de estos objetos con las mismas propiedades y métodos, tenemos que usar constructores. Un constructor es una función anónima que define un nuevo objeto y lo devuelve, creando copias del objeto (también llamadas instancias), cada una con sus propias propiedades, métodos y valores.

```
var constructor = function() {
  var obj = {
    nombre: "Juan",
    edad: 30,
    mostrarnombre: function() {
      alert(this.nombre);
    },
    cambiarnombre: function(nombrenuevo) {
      this.nombre = nombrenuevo;
    }
  };
  return obj;
};
var empleado = constructor();
empleado.mostrarnombre();  // "Juan"
```

Listado 4-74: *Usando un constructor para crear un objeto*

En el ejemplo del Listado 4-74 se asigna una función anónima a la variable constructor. Dentro de la función, se crea un objeto y se devuelve mediante la instrucción return. Finalmente, el objeto que devuelve la función se almacena en la variable empleado y se ejecuta su método mostrarnombre().

Usando constructores, podemos crear nuevos objetos de forma dinámica. Por ejemplo, podemos configurar valores iniciales para las propiedades enviando los valores a la función cuando se construye el objeto.

```
var constructor = function(nombreinicial) {
  var obj = {
    nombre: nombreinicial,
    edad: 30,
    mostrarnombre: function() {
      alert(this.nombre);
    },
    cambiarnombre: function(nombrenuevo) {
      this.nombre = nombrenuevo;
    }
  };
  return obj;
};
var empleado = constructor("Juan");
empleado.mostrarnombre();  // "Juan"
```

Listado 4-75: *Enviando valores iniciales al constructor*

El propósito de un constructor es el de funcionar como una fábrica de objetos. El siguiente ejemplo ilustra cómo crear múltiples objetos con el mismo constructor.

```
var constructor = function(nombreinicial) {
  var obj = {
    nombre: nombreinicial,
    edad: 30,
    mostrarnombre: function() {
      alert(this.nombre);
    },

    cambiarnombre: function(nombrenuevo) {
      this.nombre = nombrenuevo;
    }
  };
  return obj;
};
var empleado1 = constructor("Juan");
var empleado2 = constructor("Roberto");
var empleado3 = constructor("Arturo");
alert(empleado1.nombre + ", " + empleado2.nombre + ", " +
empleado3.nombre);
```

Listado 4-76: *Usando constructores para crear múltiples objetos*

Aunque los objetos creados desde un constructor comparten las mismas propiedades y métodos, se almacenan en diferentes espacios de memoria y, por lo tanto, manipulan valores diferentes. Cada vez que se llama la función constructor, se crea un nuevo objeto y podemos almacenar diferentes valores en cada uno de ellos. En el ejemplo del Listado 4-76, hemos creado tres objetos: empleado1, empleado2, y empleado3, y se ha asignado los valores "Juan", "Roberto", y "Arturo" a sus propiedades **`nombre`**. En consecuencia, cuando leemos la propiedad **`nombre`** de cualquiera de estos objetos, obtenemos diferentes valores dependiendo del objeto al que pertenece la propiedad (la función **`alert()`** al final del código muestra el mensaje "Juan, Roberto, Arturo").

Una ventaja de usar constructores para crear objetos es la posibilidad de definir propiedades y métodos privados. Todo los objetos que hemos creado hasta el momento contienen propiedades y métodos públicos, lo cual significa que se puede acceder a sus contenidos y modificarlos desde cualquier parte del código. Para evitar esto último y hacer que las propiedades y métodos solo sean accesibles mediante el objeto que los ha creado, tenemos que volverlos privados usando una técnica llamada closure.

Como hemos explicado anteriormente, se puede acceder a las variables creadas en el ámbito global desde cualquier lugar del código, mientras que a las variables creadas dentro de funciones solo se puede acceder desde las funciones en las que se han creado. Lo que no mencionamos es que las funciones y, por lo tanto, los métodos mantienen un enlace que las conecta al ámbito en el que se han creado y quedan conectadas a las variables declaradas en ese ámbito. Cuando devolvemos un objeto desde un constructor, sus métodos aún pueden acceder a las variables de la función, incluso cuando ya no se encuentran en el mismo ámbito, y por ello estas variables se vuelven accesibles solo para el objeto.

```
var constructor = function() {
  var nombre = "Juan";
  var edad = 30;
  var obj = {
    mostrarnombre: function() {
        alert(nombre);
    },
    cambiarnombre: function(nombrenuevo) {
      nombre = nombrenuevo;
    }
  };
  return obj;
};
var empleado = constructor();
empleado.mostrarnombre();  // "Juan"
```

Listado 4-77: *Definiendo propiedades privadas*

El código del Listado 4-77 es exactamente el mismo que hemos definido en el ejemplo anterior excepto que en lugar de declarar nombre y edad como propiedades del objeto, las declaramos como variables de la función que está devolviendo el objeto. El objeto devuelto recordará estas variables, pero el mismo será el único que tendrá acceso a ellas. No existe forma de modificar los valores de esas variables desde otras instrucciones en el código que no sea por medio de los métodos que devuelve la función (en este caso, mostrarnombre() y cambiarnombre()). Esta es la razón por la que el procedimiento se denomina closure (clausura). La función se cierra y no se puede acceder a su ámbito, pero mantenemos un elemento conectado a ella (un objeto en nuestro ejemplo).

Los métodos en este ejemplo acceden a las variables sin usar la palabra clave this. Esto se debe a que los valores ahora se almacenan en variables definidas por la función y no en propiedades definidas por el objeto.

Lo básico: cada nuevo objeto creado por un constructor se almacena en un espacio diferente en la memoria y, por lo tanto, tiene la misma estructura y sus propias variables privadas y valores, pero también podemos asignar el mismo objeto a distintas variables. Si quiere asegurarse de que una variable no está referenciando al mismo objeto, puede comparar las variables con los operadores especiales `===` y `¡==`. JavaScript también incluye el método `is()` dentro de un objeto global llamado `Object` que podemos usar para comprobar si dos variables referencian el mismo objeto (por ejemplo, `Object.is(objeto1, objeto2)`).

3.6. El operador new

Con la notación literal y los constructores tenemos todo lo que necesitamos para crear objetos individuales o múltiples objetos basados en una misma definición, pero para ser coherente con otros lenguajes de programación orientada a objetos, JavaScript ofrece una tercera alternativa. Se trata de una clase especial de constructor que trabaja con un operador llamado new (nuevo). El objeto se define mediante una función y luego se llama con el operador new para crear un objeto a partir de esa definición.

```
function MiObjeto() {
  this.nombre = "Juan";
  this.edad = 30;
  this.mostrarnombre = function(){
    alert(this.nombre);
  };
  this.cambiarnombre = function(nombrenuevo){
    this.nombre = nombrenuevo;
  };
}
var empleado = new MiObjeto();
empleado.mostrarnombre();  // "Juan"
```

Listado 4-78: *Creando objetos con el operador* `new`

Estos tipos de constructores requieren que las propiedades y los métodos de los objetos sean identificados mediante la palabra clave this, pero excepto por este requisito, la definición de estos constructores y los que hemos estudiado anteriormente son iguales.

También podemos proveer valores iniciales, como en el siguiente ejemplo.

```
function MiObjeto(nombreinicial, edadinicial){
  this.nombre = nombreinicial;
  this.edad = edadinicial;
  this.mostrarnombre = function(){
    alert(this.nombre);
  };
  this.cambiarnombre = function(nombrenuevo){
    this.nombre = nombrenuevo;
  };
}
var empleado = new MiObjeto("Roberto", 55);
empleado.mostrarnombre();  // "Roberto"
```

Listado 4-79: *Definiendo valores iniciales para el objeto*

3.7. Herencia

Una característica importante de los objetos es que podemos crearlos desde otros objetos. Cuando los objetos se crean a partir de otros objetos, pueden heredar sus propiedades y métodos, y también agregar los suyos propios. La herencia en JavaScript (cómo los objetos obtienen las mismas propiedades y métodos de otros objetos) se logra a través de prototipos. Un objeto no hereda las propiedades y los métodos de otro objeto directamente; lo hace desde el prototipo del objeto. Trabajando con prototipos puede resultar muy confuso, pero JavaScript incluye el método 279áximo() para simplificar nuestro trabajo. Este método es parte de un objeto global predefinido por JavaScript llamado Object. El método utiliza un objeto que ya existe como prototipo de uno nuevo, de modo que podemos crear objetos a partir de otros objetos sin preocuparnos de cómo se comparten entre ellos las propiedades y los métodos.

```
var miobjeto = {
  nombre: "Juan",
  edad: 30,
  mostrarnombre: function(){
    alert(this.nombre);
  },
  cambiarnombre: function(nombrenuevo){
    this.nombre = nombrenuevo;
  }
};
var empleado = Object.create(miobjeto);
empleado.cambiarnombre('Roberto');
empleado.mostrarnombre();  // "Roberto"
miobjeto.mostrarnombre();  // "Juan"
```

Listado 4-80: *Creando objetos a partir de otros objetos*

El código del Listado 4-80 crea el objeto miobjeto usando notación literal y luego llama al método 279áximo() para crear un nuevo objeto basado en el objeto miobjeto. El método áximo() solo requiere el nombre del objeto que se va usar como prototipo del nuevo objeto, y devuelve este nuevo objeto que podemos asignar a una variable para su uso posterior. En este ejemplo, el nuevo objeto se crea con Object.create() y luego se asigna a la variable empleado. Una vez que tenemos el nuevo objeto, podemos actualizar sus valores. Usando el método cambiarnombre(), cambiamos el nombre de empleado a "Roberto" y luego mostramos el valor de la propiedad nombre de cada objeto en la pantalla.

Este código crea dos objetos independientes, miobjeto y empleado, con sus propias propiedades, métodos y valores, pero conectados a través de la cadena de prototipos. El nuevo objeto empleado no es solo una copia del original, es un objeto que mantiene un enlace con el prototipo de miobjeto. Cuando introducimos cambios a este prototipo, los objetos siguientes en la cadena heredan estos cambios.

```
var miobjeto = {
  nombre: "Juan",
  edad: 30,
  mostrarnombre: function(){
    alert(this.nombre);
  },

cambiarnombre: function(nombrenuevo){
    this.nombre = nombrenuevo;
  }
};
var empleado = Object.create(miobjeto);
empleado.edad = 24;

miobjeto.mostraredad = function(){
  alert(this.edad);
};
empleado.mostraredad();  // 24
```

Listado 4-81: *Agregando un nuevo método al prototipo*

En el Listado 4-81, se agrega un método llamado **mostraredad()** al objeto prototipo (**miobjeto**). Debido a la cadena de prototipos, este nuevo método es accesible también desde las otras instancias. Cuando llamamos al método **mostraredad()** de **empleado** al final del código, el intérprete busca el método primero en **empleado** y continúa buscando hacia arriba en la cadena de prototipos hasta que lo encuentra en el objeto **miobjeto**. Cuando finalmente se encuentra el método, muestra el valor 24 en la pantalla. Esto se debe a que, a pesar de que el método **mostraredad()** se ha definido dentro de **miobjeto**, la palabra clave **this** en este método apunta al objeto con el que estamos trabajando (**empleado**). Debido a la cadena de prototipos, se puede invocar al método **mostraredad()** desde **empleado**, y debido a la palabra clave **this**, el valor de la propiedad **edad** que se muestra en la pantalla es el que se ha asignado a **empleado**.

El método **280áximo()** es tan sencillo como poderoso: toma un objeto y lo convierte en el prototipo de uno nuevo. Esto nos permite construir una cadena de objetos donde cada uno hereda las propiedades y los métodos de su predecesor.

```
var miobjeto = {
  nombre: "Juan",
  edad: 30,
  mostrarnombre: function(){
    alert(this.nombre);
  },
  cambiarnombre: function(nombrenuevo){
    this.nombre = nombrenuevo;
  }
};
var empleado1 = Object.create(miobjeto);
var empleado2 = Object.create(empleado1);
var empleado3 = Object.create(empleado2);

empleado2.mostraredad = function(){
  alert(this.edad);
};
empleado3.edad = 24;
empleado3.mostraredad();  // 24
```

Listado 4-82: *Probando la cadena de prototipos*

En el Listado 4-82, la existencia de la cadena de prototipos se demuestra agregando el método mostraredad() a empleado2. Ahora, empleado2 y empleado3 (y también cualquier otro objeto creado posteriormente a partir de estos dos objetos) tienen acceso a este método, pero debido a que la herencia funciona hacia abajo de la cadena y no hacia arriba, el método no está disponible para el objeto empleado1.

IMPORTANTE: si se acostumbra a trabajar con el método **281áximo()**, no necesitará acceder y modificar los prototipos de los objetos directamente. De hecho, el método **281áximo()** debería ser la forma estándar de trabajar en JavaScript, creando objetos sin tener que lidiar con la complejidad de su sistema de prototipos. Sin embargo, los prototipos son la esencia de este lenguaje y en algunas circunstancias no podemos evitarlos. Para obtener más información sobre los prototipos y cómo trabajar con objetos, visite nuestro sitio web y siga los enlaces de este capítulo.

Capítulo 4
Objetos estándar

4.1. Objetos estándar

Los objetos son como envoltorios de código y JavaScript se aprovecha de esta característica extensamente. De hecho, casi todo en JavaScript es un objeto. Por ejemplo, los números y las cadenas de caracteres que asignamos a las variables se convierten automáticamente en objetos por el intérprete JavaScript. Cada vez que asignamos un nuevo valor a una variable, en realidad estamos asignando un objeto que contiene ese valor.

Debido a que los valores que almacenamos en variables son de tipos diferentes, existen diferentes tipos de objetos disponibles para representarlos. Por ejemplo, si es valor es una cadena de caracteres, el objeto creado para almacenarlo es del tipo String. Cuando asignamos un texto a una variable, JavaScript crea un objeto String, almacena la cadena de caracteres en el objeto y asigna ese objeto a la variable. Si queremos, podemos crear estos objetos directamente usando sus constructores. Existen diferentes constructores disponibles dependiendo del tipo de valor que queremos almacenar. Los siguientes son los más usados.

Number(valor**)**—Este constructor crea objetos para almacenar valores numéricos. Acepta números y también cadenas de caracteres con números. Si el valor especificado por el atributo no se puede convertir en un número, el constructor devuelve el valor **`NaN`** (No es un Número).

String(valor**)**—Este constructor crea objetos para almacenar cadenas de caracteres. Acepta una cadena de caracteres o cualquier valor que pueda convertirse en una cadena de caracteres, incluidos números.

Boolean(valor**)**—Este constructor crea objetos para almacenar valores booleanos. Acepta los valores **`true`** y **`false`**. Si el valor se omite o es igual a 0, **`NaN`**, **`null`**, **`undefined`**, o una cadena de caracteres vacía, el valor almacenado en el objeto es **`false`**, en caso contrario es **`true`**.

Array(valores**)**—Este constructor crea objetos para almacenar arrays. Si se proveen múltiples valores, el constructor crea un array con esos valores, pero si solo se provee un valor, y ese valor es un número entero, el constructor crea un array con la cantidad de elementos que indica el valor del atributo y almacena el valor **`undefined`** en cada índice.

Estos constructores trabajan con el operador new para crear nuevos objetos. El siguiente ejemplo almacena un número.

```
var valor = new Number(5);
alert(valor);  // 5
```

Listado 4-83: *Creando números con un constructor*

La ventaja de usar constructores es que pueden procesar diferentes tipos de valores. Por ejemplo, podemos obtener un número a partir de una cadena de caracteres que contiene un número.

```
var valor = new Number("5");
alert(valor);  // 5
```

Listado 4-84: *Creando números a partir de cadenas de caracteres*

La cadena de caracteres que provee al constructor Number() en el Listado 4-84 se convierte en un valor numérico y se almacena en un objeto Number. Debido a que JavaScript se encarga de convertir estos objetos en valores primitivos y viceversa, podemos realizar operaciones aritméticas sobre el valor almacenado en el objeto, como lo hacemos con cualquier otro valor numérico.

```
var valor = new Number("5");
var total = valor * 35;
alert(total);  // 175
```

Listado 4-85: *Realizando operaciones aritméticas con objetos*

El constructor Array() se comporta de un modo diferente. Si especificamos varios valores, el array se crea como si lo hubiéramos declarado con corchetes.

```
var lista = new Array(12, 35, 57, 8);
alert(lista);  // 12,35,57,8
```

Listado 4-86: *Creando un array con un constructor*

Por otro lado, si especificamos un único valor y ese valor es un número entero, el constructor lo utiliza para determinar el tamaño del array, crea un array con esa cantidad de elementos y asigna el valor undefined a cada uno de ellos.

```
var lista = new Array(2);
alert(lista[0] + " - " + lista[1]);  // undefined - undefined
```

Listado 4-87: *Creando un array vacío con un constructor*

4.2. Objetos String

Convertir valores en objetos permite al lenguaje ofrecer la funcionalidad necesaria para manipular esos valores. Las siguientes son las propiedades y los métodos más usados de los objetos String.

Length—Esta propiedad devuelve un entero que representa la cantidad de caracteres.

toLowerCase()—Este método convierte los caracteres a letras minúsculas.

toUpperCase()—Este método convierte los caracteres a letras mayúsculas.

Trim()—Este método elimina espacios en blanco a ambos lados de la cadena de caracteres. JavaScript también incluye los métodos **`trimLeft()`** y **`trimRight()`** para limpiar la cadena de caracteres en un lado específico (izquierda o derecha).

Substr(comienzo, extensión**)**—Este método devuelve una nueva cadena de caracteres con caracteres extraídos de la cadena original. El atributo **comienzo** indica la posición del primer carácter a leer, y el atributo **extensión** determina cuántos caracteres queremos incluir. Si no se especifica la extensión, el método devuelve todos los caracteres hasta el final de la cadena.

Substring(comienzo, final**)**—Este método devuelve una nueva cadena de caracteres con caracteres extraídos de la cadena original. Los atributos **comienzo** y **final** son números enteros que determinan las posiciones del primer y último carácter a incluir.

startsWith(valor**)**—Este método devuelve **`true`** si el texto especificado por el atributo **valor** se encuentra al comienzo de la cadena de caracteres.

endsWith(valor**)**—Este método devuelve **`true`** si el texto especificado por el atributo **valor** se encuentra al final de la cadena de caracteres.

Includes(buscar, posición**)**—Este método busca el valor del atributo **buscar** dentro de la cadena y devuelve **`true`** o **`false`** de acuerdo con el resultado. El atributo **buscar** es el texto que queremos buscar, y el atributo **posición** determina el índice en el que queremos comenzar la búsqueda. Si el atributo **posición** no se especifica, la búsqueda comienza desde el inicio de la cadena.

indexOf(valor, posición**)**—Este método devuelve el índice en el que el texto especificado por el atributo **valor** se encuentra por primera vez. El atributo **posición** determina el índice en el que queremos comenzar la búsqueda. Si el atributo **posición** no se especifica, la búsqueda comienza desde el inicio de la cadena. El método devuelve el valor -1 si ninguna coincidencia es encontrada.

lastIndexOf(valor, posición**)**—Este método devuelve el índice en el que se encuentra por primera vez el texto especificado por el atributo **valor**. A diferencia de **`indexOf()`**, este método realiza una búsqueda hacia atrás, desde el final de la cadena. El atributo **posición** determina el índice en el que queremos comenzar la búsqueda. Si no se especifica el atributo **posición**, la búsqueda comienza desde el final de la cadena. El método devuelve el valor -1 si no se encuentra ninguna coincidencia.

Replace(expresión, reemplazo**)**—Este método reemplaza la parte de la cadena de caracteres que coincide con el valor del atributo **expresión** por el texto especificado por el atributo **reemplazo**. El atributo **expresión** se puede especificar como una cadena de caracteres o como una expresión regular para buscar un texto con un formato particular.

Las cadenas de caracteres se almacenan como arrays de caracteres, de modo que podemos acceder a cada carácter usando corchetes, como lo hacemos con cualquier otro array. JavaScript incluye la propiedad length para contar la cantidad de caracteres en la cadena.

```
var texto = "Hola Mundo";
var mensaje = "El texto tiene " + texto.length + " caracteres";
alert(mensaje);  // "El texto tiene 10 caracteres"
```

Listado 4-88: *Contando la cantidad de caracteres en una cadena*

Debido a que las cadenas de caracteres se almacenan como arrays, podemos iterar sobre los caracteres con un bucle. En el siguiente ejemplo, agregamos un espacio entre las letras de un texto.

```
var texto = "Hola Mundo";
var mensaje = "";
for (var f = 0; f < texto.length; f++) {
  mensaje += texto[f] + " ";
}
alert(mensaje);  // "H o l a   M u n d o "
```

Listado 4-89: *Iterando sobre los caracteres de una cadena*

La propiedad length devuelve el número de caracteres en la cadena, pero los índices comienzan a contar desde 0, por lo que debemos crear un bucle que vaya desde 0 al valor anterior de la propiedad length para obtener todos los caracteres en la cadena. Usando estos índices, el bucle for del ejemplo del Listado 4-89 lee cada carácter usando corchetes y los agrega al valor actual de la variable mensaje junto con un espacio. En consecuencia, obtenemos una nueva cadena de caracteres con todos los caracteres de la cadena original separados por un espacio.

Este ejemplo agrega un espacio después de cada carácter en la cadena, lo cual significa que el texto final termina con un espacio en blanco. Los objetos String ofrecen el método trim() para eliminar estos espacios no deseados.

```
var texto = "Hola Mundo";
var mensaje = "";
for (var f = 0; f < texto.length; f++) {
  mensaje += texto[f] + " ";
}
mensaje = mensaje.trim();
alert(mensaje);  // "H o l a   M u n d o"
```

Listado 4-90: *Removiendo espacios*

La posibilidad de acceder a cada carácter en una cadena nos permite lograr efectos interesantes. Solo tenemos que detectar la posición del carácter que queremos manipular y realizar los cambios que deseamos. El siguiente ejemplo agrega puntos entre las letras de cada palabra, pero no entre las palabras.

```
var texto = "Hola Mundo";
var mensaje = "";
var anterior = "";

for (var f = 0; f < texto.length; f++) {
  if (mensaje != "") {
    anterior = texto[f - 1];
    if (anterior != " " && texto[f] != " ") {
      mensaje += ".";
    }
  }
  mensaje += texto[f];
}
alert(mensaje);  // "H.o.l.a M.u.n.d.o"
```

Listado 4-91: *Procesando una cadena de caracteres*

El código del Listado 4-91 comprueba si el carácter que estamos leyendo y el que hemos leído en el ciclo anterior no son espacios en blanco antes de agregar un punto al valor actual de la variable mensaje. De este modo, los puntos se insertan solo entre las letras y no al final o comienzo de cada palabra.

Debido a la complejidad que pueden alcanzar algunos de los procesos realizados con cadenas de caracteres, JavaScript incluye varios métodos para simplificar nuestro trabajo. Por ejemplo, podemos reemplazar todos los caracteres en una cadena con letras mayúsculas con solo llamar al método toUpperCase().

```
var texto = "Hola Mundo";
var mensaje = texto.toUpperCase();
alert(mensaje);  // "HOLA MUNDO"
```

Listado 4-92: *Convirtiendo los caracteres de una cadena en letras mayúsculas*

A veces no necesitamos trabajar con todo el texto, sino solo con algunas palabras o caracteres. Los objetos String incluyen los métodos substr() y substring() para copiar un trozo de texto desde una cadena. El método substr() copia el grupo de caracteres que comienza en el índice especificado por el primer atributo. También se puede especificar un segundo atributo para determinar cuántos caracteres queremos incluir desde la posición inicial.

```
var texto = "Hola Mundo";
var palabra = texto.substr(0, 4);
alert(palabra);  // "Hola"
```

Listado 4-93: *Copiando un grupo de caracteres*

El método substr() del Listado 4-93 copia un total de cuatro caracteres comenzando con el carácter en el índice 0. Como resultado, obtenemos la cadena "Hola". Si no especificamos el número de caracteres a incluir, el método devuelve todos los caracteres hasta que llega al final de la cadena.

```
var texto = "Hola Mundo";
var palabra = texto.substr(5);
alert(palabra);  // "Mundo"
```

Listado 4-94: *Copiando todos los caracteres desde un índice hasta el final de la cadena*

El método substr() también puede tomar valores negativos. Cuando se especifica un índice negativo, el método cuenta de atrás hacia adelante. El siguiente código copia los mismos caracteres que el ejemplo anterior.

```
var texto = "Hola Mundo";
var palabra = texto.substr(-5);
alert(palabra);  // "Mundo"
```

Listado 4-95: *Referenciando caracteres con índices negativos*

El método substring() trabaja de una forma diferente al método substr(). Este método toma dos valores para determinar los caracteres primero y último que queremos copiar, pero no incluye el último carácter.

```
var texto = "Hola Mundo";
var palabra = texto.substring(5, 7);
alert(palabra);  // "Mu"
```

Listado 4-96: *Copiando caracteres entre dos índices*

El método substring() del Listado 4-96 copia los caracteres desde el índice 5 al 7 de la cadena (el carácter en el último índice no se incluye). Estas son las posiciones en las que se encuentran los caracteres "Mu" y, por lo tanto, este es el valor que obtenemos en respuesta.

Si lo que necesitamos es extraer las palabras de una cadena, podemos usar el método split(). Este método corta la cadena en partes más pequeñas y devuelve un array con estos valores. El método requiere un valor con el carácter que queremos usar como separador, por lo que si usamos un espacio, podemos dividir la cadena en palabras.

```
var texto = "Hola Mundo";
var palabras = texto.split(" ");
alert(palabras[0] + " / " + palabras[1]);  // "Hola / Mundo"
```

Listado 4-97: *Obteniendo las palabras de una cadena*

El método split() del Listado 4-97 crea dos cadenas de caracteres con las palabras "Hola" y "Mundo", y devuelve un array con estos valores, que podemos leer como hacemos con los valores de cualquier otro array.

A veces no sabemos dónde se encuentran los caracteres que queremos modificar o si la cadena incluye esos caracteres. Existen varios métodos disponibles en los objetos String para hacer una búsqueda. El que usemos depende de lo que queremos lograr. Por ejemplo, los métodos startsWith() y endsWith() buscan un texto al comienzo o al final de la cadena, y devuelven true si se encuentra el texto.

```
var texto = "Hola Mundo";
if (texto.startsWith("Ho")) {
  alert("El texto comienza con 'Ho'");
}
```

Listado 4-98: *Buscando un texto al comienzo de la cadena*

Debido a que estos métodos devuelven valores booleanos, podemos usarlos para establecer la condición de una instrucción if. En el código del Listado 4-98, buscamos el

texto “Ho” al comienzo de la variable texto y mostramos un mensaje en caso de éxito. En este ejemplo, se encuentra el texto, el método devuelve true y, por lo tanto, el mensaje se muestra en pantalla.

Si el texto que estamos buscando puede encontrarse en cualquier parte de la cadena, no solo al comienzo o al final, podemos usar el método includes(). Al igual que los métodos anteriores, el método includes() busca un texto y devuelve true en caso de éxito, pero la búsqueda se realiza en toda la cadena.

```
var texto = "Hola Mundo";
if (texto.includes("l")) {
  alert("El texto incluye la letra L");
}
```

Listado 4-99: *Buscando un texto dentro de otro texto*

Hasta el momento, hemos comprobado si uno o más caracteres se encuentran dentro de una cadena, pero a veces necesitamos saber dónde se han encontrado esos caracteres. Existen dos métodos para este propósito: indexOf() y lastIndexOf(). Ambos métodos devuelven el índice donde se encuentra la primera coincidencia, pero el método indexOf() comienza la búsqueda desde el inicio de la cadena y el método lastIndexOf() lo hace desde el final.

```
var texto = "Hola Mundo";
var 288áximo = texto.indexOf("Mundo");
alert("La palabra comienza en el índice " + 288áximo);  // 5
```

Listado 4-100: *Encontrando la ubicación de un texto dentro de una cadena de caracteres*

Una vez que conocemos la posición de los caracteres que estamos buscando, podríamos crear un pequeño código que reemplace esos caracteres por otros diferentes, pero esto no es necesario. Los objetos String incluyen un método llamado replace() con este propósito. Este es un método complejo que puede tomar múltiples valores y trabajar no solo con cadenas de caracteres, sino con expresiones regulares, pero también podemos usarlo con textos breves o palabras. El siguiente ejemplo reemplaza la palabra “Mundo” por la palabra “Planeta”.

```
var texto = "Hola Mundo";
var textonuevo = texto.replace("Mundo", "Planeta");
alert(textonuevo);  // "Hola Planeta"
```

Listado 4-101: *Reemplazando textos en una cadena de caracteres*

4.3. Objetos Array

Como ya mencionamos, en JavaScript los arrays también son objetos e incluyen propiedades y métodos para manipular sus valores. Los siguientes son los más usados.

Length—Esta propiedad devuelve un número entero que representa la cantidad de valores en el array.

Push(valores**)**—Este método agrega uno o más valores al final del array y devuelve la nueva extensión del array. También contamos con un método similar llamado **`unshift()`**, que agrega los valores al comienzo del array.

Pop()—Este método elimina el último valor del array y lo devuelve. También contamos con un método similar llamado **`shift()`**, que elimina el primer valor del array.

Concat(array**)**—Este método concatena el array con el array especificado por el atributo y devuelve un nuevo array con el resultado. Los arrays originales no se modifican.

Splice(índice, remover, valores**)**—Este método agrega o elimina valores de un array y devuelve un nuevo array con los elementos eliminados. El atributo **índice** indica el índice en el que vamos a introducir las modificaciones, el atributo **remover** es el número de valores que queremos eliminar desde el índice, y el atributo **valores** es la lista de valores separados por coma que queremos agregar al array desde el índice. Si solo queremos agregar valores, el atributo **remover** se puede declarar con el valor 0, y si solo queremos eliminar valores, podemos ignorar el último atributo.

Slice(comienzo, final**)**—Este método copia los valores en las posiciones indicadas por los atributos dentro de un nuevo array. Los atributos **comienzo** y **final** indican los índices del primer y último valor a copiar. El último valor no se incluye en el nuevo array.

indexOf(valor, posición**)**—Este método devuelve el índice en el que se encuentra por primera vez el valor especificado por el atributo **valor**. El atributo **posición** determina el índice en el que queremos comenzar la búsqueda. Si el atributo **posición** no se especifica, la búsqueda comienza desde el inicio del array. El método devuelve el valor -1 si no se encuentra ninguna coincidencia.

lastIndexOf(valor, posición**)**—Este método devuelve el índice en el que el valor especificado por el atributo **valor** se encuentra por primera vez. A diferencia de **`indexOf()`**, este método realiza una búsqueda de atrás hacia adelante. El atributo **posición** determina el índice en el que queremos comenzar la búsqueda. Si no se especifica el atributo **posición**, la búsqueda comienza desde el final del array. El método devuelve el valor -1 si no se encuentra ninguna coincidencia.

Filter(función**)**—Este método envía los valores del array a una función uno por uno y devuelve un nuevo array con todos los valores que aprueba la función. El atributo **función** es una referencia a una función o una función anónima a cargo de validar los valores. La función recibe tres valores: el valor a evaluar, su índice y una referencia al array. Después de procesar el valor, la función debe devolver un valor booleano para indicar si es válido o no.

every(función**)**—Este método envía los valores del array a una función uno por uno y devuelve **`true`** cuando la función aprueba todos los valores. El atributo **función** es una referencia a una función o una función anónima a cargo de evaluar los valores. La función recibe tres valores: el valor a evaluar, su índice, y una referencia al array. Después de procesar el valor, la función debe devolver un valor booleano indicando si es válido o no. También contamos con un método similar llamado **`some()`** que devuelve **`true`** si la función aprueba al menos uno de los valores.

Join(separador**)**—Este método crea una cadena de caracteres con todos los valores en el array. El atributo **separador** especifica el carácter o la cadena de caracteres que queremos incluir entre los valores.

Reverse()—Este método invierte el orden de los valores en el array.

Sort(función**)**—Este método ordena los valores en el array. El atributo **función** es una referencia a una función o una función anónima a cargo de comparar los valores. La función recibe dos valores desde el array y debe devolver un valor booleano indicando su orden. Si no se especifica el atributo, el método ordena los elementos alfabéticamente y en orden ascendente.

Map(función**)**—Este método envía los valores del array a una función uno por uno y crea un nuevo array con los valores que devuelve la función. El atributo **`función`** es una referencia a una función o una función anónima a cargo de procesar los valores. La función recibe tres valores: el valor a procesar, su índice y una referencia al array.

Al igual que los objetos String, los objetos Array ofrecen la propiedad length para obtener la cantidad de valores en el array. La implementación es la misma; solo tenemos que leer la propiedad para obtener el valor en respuesta.

```
var lista = [12, 5, 80, 34];
alert(lista.length);  // 4
```

Listado 4-102: *Obteniendo la cantidad de valores en el array*

Usando esta propiedad, podemos iterar sobre el array con un bucle for, como hemos hecho anteriormente con cadenas de caracteres (ver Listado 4-89). El valor que devuelve la propiedad se usa para definir la condición del bucle.

```
var lista = [12, 5, 80, 34];
var total = 0;
for (var f = 0; f < lista.length; f++) {
  total += lista[f];
}
alert("El total es: " + total);  // "El total es: 131"
```

Listado 4-103: *Iterando sobre el array*

En el código del Listado 4-103, el valor que devuelve la propiedad length es 4, de modo que el bucle va de 0 a 3. Usando estos valores dentro del bucle, podemos leer los valores

del array y procesarlos.

Aunque podemos iterar sobre el array para leer y procesar los valores uno por uno, los objetos Array ofrecen otros métodos para acceder a ellos. Por ejemplo, si queremos procesar solo algunos de los valores en el array, podemos obtener una copia con el método slice().

```
var lista = [12, 5, 80, 34];
var listanueva = lista.slice(1, 3);
alert(listanueva);  // 5,80
```

Listado 4-104: *Creando un array con los valores de otro array*

El método slice() devuelve un nuevo array con los valores entre el índice especificado por el primer atributo y el valor en el índice que se encuentra antes del especificado por el segundo atributo. En el ejemplo del Listado 4-104, el método accede a los valores en los índices 1 y 2, y devuelve los números 5 y 80.

Si queremos examinar los valores antes de incluirlos en el nuevo array, tenemos que usar un filtro. Con este propósito, los objetos Array ofrecen el método filter(). Este método envía cada valor a la función y los incluye en el nuevo array solo cuando la función devuelve true. En el siguiente ejemplo, devolvemos true cuando el valor es menor o igual que 50. En consecuencia, el nuevo array contiene todos los valores del array original excepto el valor 80.

```
var lista = [12, 5, 80, 34];
var listanueva = lista.filter(function(valor) {
  if (valor <= 50) {
    return true;
  }  else  {
    return false;
  }});
alert(listanueva);  // 12, 5, 34
```

Listado 4-105: *Filtrando los valores de un array*

En la función provista para el método filter() del Listado 4-105, comparamos el valor del array con el número 50 y devolvemos true o false dependiendo de la condición, pero como las condiciones ya producen un valor booleano cuando se evalúan, podemos devolver la condición misma y simplificar el código.

```
var lista = [12, 5, 80, 34];
var listanueva = lista.filter(function(valor) {
  return valor <= 50;
});
alert(listanueva);  // 12, 5, 34
```

Listado 4-106: *Devolviendo una condición para filtrar los elementos*

Parecidos a filter() son los métodos every() y some(). Estos métodos evalúan los valores con una función, pero en lugar de devolver un array con los valores validados por la función, devuelven los valores true o false. El método every() devuelve true si se validan todos los valores, y el método some() devuelve true si se valida al menos uno de los valores. El siguiente ejemplo usa la función every() para comprobar que todos los valores del array son menores o iguales a 100.

```
var lista = [12, 5, 80, 34];
var valido = lista.every(function(valor) {
  return valor <= 100;
});
if (valido) {
  alert("Los valores no son mayores que 100");
}
```

Listado 4-107: *Evaluando los valores de un array*

Si lo que queremos es incluir los valores de una array dentro de una cadena de caracteres para mostrarlos al usuario, podemos llamar al método join(). Este método crea una cadena de caracteres con los valores del array separados por un carácter o una cadena de caracteres. El siguiente ejemplo crea una cadena de caracteres con un guion entre los valores.

```
var lista = [12, 5, 80, 34];
var mensaje = lista.join("-");
alert(mensaje);  // "12-5-80-34"
```

Listado 4-108: *Creando una cadena de caracteres con los valores de un array*

Otra manera de acceder a los valores de un array es con los métodos indexOf() y lastIndexOf(). Estos métodos buscan un valor y devuelven el índice de la primer coincidencia encontrada. El método **`indexOf()`** comienza la búsqueda desde el inicio delarray y el método **`lastIndexOf()`** lo hace desde el final.

```
var lista = [12, 5, 80, 34, 5];
var 292áximo = lista.indexOf(5);
alert("El valor " + lista[292áximo] + " se encuentra en el índice " +
292áximo);
```

Listado 4-109: *Obteniendo el índice de un valor en el array*

El código del Listado 4-109 busca el valor 5 en el array y devuelve el índice 1. Esto se debe a que el método solo devuelve el índice del primer valor que coincide con la búsqueda. Si queremos encontrar todos los valores que coinciden con la búsqueda, tenemos que crear un bucle para realizar múltiples búsquedas, como muestra el siguiente ejemplo.

```
var lista = [12, 5, 80, 34, 5];
var buscar = 5;
var ultimo = 0;
var contador = 0;
while (ultimo >= 0) {
  var ultimo = lista.indexOf(5, ultimo);
  if (ultimo ¡= -1) {
    ultimo++;
    contador++;
  }
}
alert("Hay un total de " + contador + " números " + buscar);
```

Listado 4-110: *Buscando múltiples valores en un array*

El método indexOf() puede tomar un segundo atributo que especifica la ubicación desde la que queremos comenzar la búsqueda. Usando este atributo, podemos indicar al método que no busque en los índices donde ya hemos encontrado un valor. En el ejemplo del Listado 4-110, usamos la variable ultimo para este propósito. Esta variable almacena el índice del último valor encontrado por el método indexOf(). Al comienzo, la variable se inicializa con el valor 0, lo cual significa que el método comenzará la búsqueda desde el índice 0 del array, pero después de realizar la primera búsqueda, la variable se actualiza con el índice que devuelve el método indexOf(), desplazando el punto de partida de la próxima búsqueda a una nueva ubicación. El bucle sigue funcionando hasta que el método indexOf() devuelve el valor -1 (no se ha encontrado ninguna coincidencia).

Además de establecer la nueva ubicación con la variable ultimo, el bucle también incrementa el valor de la variable contador para contar el número de valores encontrados. Cuando finaliza el proceso, el código muestra un mensaje en la pantalla para comunicar esta información al usuario (“Hay un total de 2 números 5”).

Hasta el momento, hemos trabajado siempre con el mismo array, pero los arrays se pueden ampliar o reducir. Para agregar valores a un array, todo lo que tenemos que hacer es llamar a los métodos push() o unshift(). El método push() agrega el valor al final del array y el método unshift() lo agrega al comienzo.

```
var lista = [12, 5, 80, 34];
lista.push(100);
alert(lista);  // 12,5,80,34,100
```

Listado 4-111: *Agregando valores a un array*

Para agregar múltiples valores, tenemos que especificarlos separados por comas.

```
var lista = [12, 5, 80, 34];
lista.push(100, 200, 300);
alert(lista);  // 12,5,80,34,100,200,300
```

Listado 4-112: *Agregando múltiples valores a un array*

Otra manera de agregar múltiples valores a un array es con el método concat(). Este método concatena dos arrays y devuelve un nuevo array con el resultado (los arrays originales no se modifican).

```
var lista = [12, 5, 80, 34];
var listanueva = lista.concat([100, 200, 300]);
alert(listanueva);  // 12,5,80,34,100,200,300
```

Listado 4-113: *Concatenando dos arrays*

Eliminar valores es también fácil de hacer con los métodos pop() y shift(). El método pop() elimina el valor al final del array y el método shift() lo elimina al inicio.

```
var lista = [12, 5, 80, 34];
lista.pop();
alert(lista);  // 12,5,80
```

Listado 4-114: *Eliminando valores de un array*

Los métodos anteriores agregan o eliminan valores al comienzo o al final del array. Si queremos más flexibilidad, podemos usar el método splice(). Este método agrega o elimina valores desde cualquier ubicación del array. El primer atributo del método splice() especifica el índice donde queremos comenzar a eliminar valores y el segundo atributo determina cuántos elementos queremos eliminar. Por ejemplo, el siguiente código elimina 2 valores desde el índice 1.

```
var lista = [12, 5, 80, 34];
var removidos = lista.splice(1, 2);
alert("Valores restantes: " + lista);  // 12,34
alert("Valores removidos: " + removidos);  // 5,80
```

Listado 4-115: *Removiendo valores de un valor*

Con este método, también podemos agregar nuevos valores a un array en posiciones específicas. Los valores se deben especificar después de los dos primeros atributos separados por comas. El siguiente ejemplo agrega los valores 24, 25, y 26 en el índice 2. Como no se va a eliminar ningún valor, declaramos el valor 0 para el segundo atributo.

```
var lista = [12, 5, 80, 34];
lista.splice(2, 0, 24, 25, 26);
alert(lista);  // 12,5,24,25,26,80,34
```

Listado 4-116: *Agregando valores a un array en una posición específica*

Los objetos Array también incluyen métodos para ordenar los valores en el array. Por ejemplo, el método reverse() invierte el orden de los valores en el array.

```
var lista = [12, 5, 80, 34];
lista.reverse();
alert(lista);  // 34,80,5,12
```

Listado 4-117: *Invirtiendo el orden de los valores de un array*

Un mejor método para ordenar arrays es sort(). Este método puede tomar una función para decidir el orden en el que se ubicarán los valores. Si no especificamos ninguna función, el método ordena el array en orden alfabético y ascendente.

```
var lista = [12, 5, 80, 34];
lista.sort();
alert(lista);  // 12,34,5,80
```

Listado 4-118: *Ordenando los valores en orden alfabético*

Los valores en este ejemplo se ordenan alfabéticamente, no de forma numérica. Si queremos que el método considere el orden numérico o lograr una organización diferente, tenemos que facilitar una función. La función recibe dos valores y debe devolver un valor booleano para indicar su orden. Por ejemplo, para ordenar los valores en orden ascendente, tenemos que devolver true si el primer valor es mayor que el segundo valor o false en caso contrario.

```
var lista = [12, 5, 80, 34];
lista.sort(function(valor1, valor2) {
  return valor1 > valor2;
});
alert(lista);  // 5,12,34,80
```

Listado 4-119: *Ordenando los valores en orden numérico*

Un método interesante que se incluye en los objetos Array es map(). Con este método podemos procesar los valores del array uno por uno y crear un nuevo array con los resultados. Al igual que otros métodos, este método envía los valores a una función, pero en lugar de un valor booleano, la función debe devolver el valor que queremos almacenar en el nuevo array. Por ejemplo, el siguiente código multiplica cada valor por 2 y devuelve los resultados.

```
var lista = [12, 5, 80, 34];
var listanueva = lista.map(function(valor) {
  return valor * 2;
});
alert(listanueva);  // 24,10,160,68
```

Listado 4-120: *Procesando los valores y almacenando los resultados*

4.4. Objetos Date

Manipular fechas es una tarea complicada, no solo porque las fechas están compuestas de varios valores que representan diferentes componentes, sino porque estos componentes están relacionados. Si un valor sobrepasa su limite, afecta al resto de los valores de la fecha. Y los límites son distintos por cada componente. El límite para minutos y segundos es 60, pero el límite para las horas es 24, y cada mes tiene un número diferente de días. Las fechas también tienen que contemplar diferentes zonas horarias, cambios de horarios por estación, etc. Para simplificar el trabajo de los desarrolladores, JavaScript define un objeto llamado Date. El objeto Date almacena una fecha y se encarga de mantener los valores dentro de sus límites. La fecha en estos objetos se almacena en milisegundos, lo cual nos permite realizar operaciones entre fechas, calcular intervalos, etc. JavaScript ofrece el siguiente constructor para crear estos objetos.

Date(valor**)**—Este constructor crea un valor en milisegundos para representar una fecha basada en los valores provistos por el atributo. El atributo **`valor`** se puede declarar como una cadena de caracteres o como los componentes de una fecha separados por comas, en este orden: año, mes, día, horas, minutos, segundos y milisegundos. Si no especificamos ningún valor, el constructor crea un objeto con la fecha actual del sistema.

La fecha almacenada en estos objetos se representa con un valor en milisegundos calculado desde el 1 de enero del año 1970. Debido a que este valor no resulta familiar para los usuarios, los objetos Date ofrecen los siguientes métodos para obtener los componentes de la fecha, como el año o el mes.

getFullYear()—Este método devuelve un número entero que representa el año (un valor de 4 dígitos).

getMonth()—Este método devuelve un número entero que representa el mes (un valor de 0 a 11).

getDate()—Este método devuelve un número entero que representa el día del mes (un valor de 1 a 31).

getDay()—Este método devuelve un número entero que representa el día de la semana (un valor de 0 a 6).

getHours()—Este método devuelve un número que representa las horas (un valor de 0 a 23).

getMinutes()—Este método devuelve un número entero que representa los minutos (un valor de 0 a 59).

getSeconds()—Este método devuelve un número entero que representa los segundos (un valor de 0 a 59).

getMilliseconds()—Este método devuelve un número entero que representa los milisegundos (un valor de 0 a 999).

getTime()—Este método devuelve un número entero en milisegundos que representa el intervalo desde el 1 de enero de 1970 hasta la fecha.

Los objetos Date también incluyen métodos para modificar los componentes de la fecha.

setFullYear(año**)**—Este método especifica el año (un valor de 4 dígitos). También puede recibir valores para especificar el mes y el día.

setMonth(mes**)**—Este método especifica el mes (un valor de 0 a 11). También puede recibir valores para especificar el día.

setDate(día**)**—Este método especifica el día (un valor de 1 a 31).

setHours(horas**)**—Este método especifica la hora (un valor de 0 a 23). También puede recibir valores para especificar los minutos y segundos.

setMinutes(minutos**)**—Este método especifica los minutos (un valor de 0 a 59). También puede recibir un valor para especificar los segundos.

setSeconds(segundos**)**—Este método especifica los segundos (un valor de 0 a 59). También puede recibir un valor para especificar los milisegundos.

setMilliseconds(milisegundos**)**—Este método especifica los milisegundos (un valor de 0 a 999).

Los objetos Date también ofrecen métodos para convertir una fecha en una cadena de caracteres.

toString()—Este método convierte una fecha en una cadena de caracteres. El valor que devuelve se expresa en inglés americano y con el formato "Wed Jan 04 2017 22:32:48 GMT-0500 (EST)".

toDateString()—Este método convierte una fecha en una cadena de caracteres, pero solo devuelve la parte de la fecha, no la hora. El valor se expresa en inglés americano y con el formato "Wed Jan 04 2017".

toTimeString()—Este método convierte una fecha en una cadena de caracteres, pero solo devuelve la hora. El valor se expresa en inglés americano y con el formato "23:21:55 GMT-0500 (EST)".

Cada vez que necesitamos una fecha, simplemente tenemos que crear un nuevo objeto con el constructor Date(). Si no especificamos una fecha, el constructor crea el objeto Date con la fecha actual del sistema.

```
var fecha = new Date();
alert(fecha);  // "Wed Jan 04 2017 20:51:17 GMT-0500 (EST)"
```

Listado 4-121: *Creando un objeto* `Date`

La fecha se puede declarar con una cadena de caracteres que contiene una fecha expresada en un formato comprensible para las personas. El constructor se encarga de convertir el texto en un valor en milisegundos.

```
var fecha = new Date("January 20 2017");
alert(fecha);  // "Fri Jan 20 2017 00:00:00 GMT-0500 (EST)"
```

Listado 4-122: *Creando un objeto* `Date` *a partir de un texto*

Debido a que los navegadores interpretan el texto de diferentes maneras, en lugar de una cadena de caracteres es recomendable crear la fecha declarando los valores de los componentes separados por comas. El siguiente ejemplo crea un objeto Date con la fecha 2017/02/15 12:35.

```
var fecha = new Date(2017, 1, 15, 12, 35, 0);
alert(fecha);  // "Wed Feb 15 2017 12:35:00 GMT-0500 (EST)"
```

Listado 4-123: *Creando un objeto* `Date` *a partir de los componentes de una fecha*

Los valores se deben declarar en el siguiente orden: año, mes, día, hora, minutos y segundos. Todos los valores son iguales a los que acostumbramos a usar para definir una fecha, pero el mes se debe representar con un número de 0 a 11, y por este motivo hemos tenido que declarar el valor 1 en el constructor para representar el mes de febrero.

Lo básico: las fechas se representan con un valor en milisegundos, pero cada vez que las mostramos en la pantalla con la función **`alert()`** el navegador automáticamente las convierte en cadenas de caracteres en un formato que el usuario puede entender. Esto lo realiza el navegador solo cuando la fecha se presenta al usuario con funciones predefinidas como **`alert()`**. Si queremos formatear la fecha en nuestro código, podemos extraer los componentes e incluirlos en una cadena de caracteres, como veremos más adelante, o llamar a los métodos provistos por los objetos **`Date`** con este propósito. El método **`toString()`** crea una cadena de caracteres con la fecha completa, el método **`toDateString()`** solo incluye los componentes de la fecha y el método **`toTimeString()`** solo incluye la hora.

Debido a que las fechas se almacenan en milisegundos, cada vez que queremos procesar los valores de sus componentes, tenemos que obtenerlos con los métodos provistos por el objeto. Por ejemplo, para obtener el año de una fecha, tenemos que usar el método getFullYear().

```
var hoy = new Date();
var ano = hoy.getFullYear();
alert("El año es " + ano);  // "El año es 2017"
```

Listado 4-124: *Leyendo los componentes de una fecha*

El resto de los componentes se obtiene de la misma manera con los métodos respectivos. Lo único que tenemos que considerar es que los meses se representan con valores de 0 a 11, y, por lo tanto, tenemos que sumar 1 al valor que devuelve el método getMonth() para obtener un valor que las personas puedan entender.

```
var hoy = new Date();
var ano = hoy.getFullYear();
var mes = hoy.getMonth() + 1;
var dia = hoy.getDate();
alert(ano + "-" + mes + "-" + dia);  // "2017-1-5"
```

Listado 4-125: *Leyendo el mes*

Estos métodos también resultan útiles cuando necesitamos incrementar o disminuir una fecha. Por ejemplo, si queremos determinar la fecha 15 días a partir de la fecha actual, tenemos que obtener el día actual, sumarle 15 y asignar el resultado al objeto.

```
var hoy = new Date();
alert(hoy);  // "Thu Jan 05 2017 14:37:28 GMT-0500 (EST)"
hoy.setDate(hoy.getDate() + 15);
alert(hoy);  // "Fri Jan 20 2017 14:37:28 GMT-0500 (EST)"
```

Listado 4-126: *Incrementando una fecha*

Si en lugar de incrementar o disminuir una fecha por un período de tiempo, lo que queremos es calcular la diferencia entre dos fechas, tenemos que restar una fecha a la otra.

```
var hoy = new Date(2017, 0, 5);
var futuro = new Date(2017, 0, 20);
var intervalo = futuro - hoy;
alert(intervalo);  // 1296000000
```

Listado 4-127: *Calculando un intervalo*

El valor que devuelve la resta se expresa en milisegundos. A partir de este valor, podemos extraer cualquier componente que necesitemos. Todo lo que tenemos que hacer es dividir el número por los valores máximos de cada componente. Por ejemplo, si queremos expresar en número en segundos, tenemos que dividirlo por 1000 (1000 milisegundos = 1 segundo).

```
var hoy = new Date(2017, 0, 5);
var futuro = new Date(2017, 0, 20);
var intervalo = futuro – hoy;
var segundos = intervalo / 1000;
alert(segundos + " segundos");  // "1296000 segundos"
```

Listado 4-128: *Calculando un intervalo en segundos*

Por supuesto, no es fácil incluir un valor en segundos dentro de un contexto que podamos entender, pero sí podemos hacerlo con un valor en días. Para expresar el intervalo en días, tenemos que seguir dividiendo el resultado. Si dividimos los milisegundos por 1000, obtenemos segundos, el resultado dividido por 60 nos da los minutos, ese resultado dividido por 60 nuevamente nos da las horas, y dividiendo el resultado por 24 obtenemos los días (24 horas = 1 día).

```
var hoy = new Date(2017, 0, 5);
var futuro = new Date(2017, 0, 20);
var intervalo = futuro – hoy;
var 299áxi = intervalo / (24 * 60 * 60 * 1000);
alert(299áxi + " días");  // "15 dias"
```

Listado 4-129: *Calculando un intervalo en días*

La comparación es otra operación útil que podemos realizar con fechas. Aunque podemos comparar objetos Date directamente, algunos operadores de comparación no comparan las fechas, sino los objetos mismos, por lo que la mejor manera de hacerlo es obtener primero las fechas en milisegundos con el método getTime() y luego comparar esos valores.

```
var hoy = new Date(2017, 0, 20, 10, 35);
var futuro = new Date(2017, 0, 20, 12, 35);

if (futuro.getTime() == hoy.getTime()) {
  alert("Las Fechas son Iguales");
} else {
  alert("Las Fechas son Diferentes");
}
```

Listado 4-130: *Comparando dos fechas*

El código del Listado 4-130 comprueba si dos fechas son iguales o no, y muestra un mensaje para comunicar el resultado. Las fechas que declaramos en este ejemplo son iguales, excepto por la hora. Una fecha se ha configurado con la hora 10:35 y la segunda fecha se ha configurado con la hora 12:35, por lo que el código determina que son diferentes. Si queremos comparar solo la fecha, sin considerar la hora, tenemos que anular los componentes de la hora. Por ejemplo, podemos configurar las hora, los minutos y segundos en ambas fechas con el valor 0 y entonces los únicos valores a comparar serán el año, el mes y el día.

```
var hoy = new Date(2017, 0, 20, 10, 35);
var futuro = new Date(2017, 0, 20, 12, 35);
hoy.setHours(0, 0, 0);
futuro.setHours(0, 0, 0);

if (futuro.getTime() == hoy.getTime()) {
  alert("Las Fechas son Iguales");
} else {
  alert("Las Fechas son Diferentes");
}
```

Listado 4-131: *Comparando solo las fechas sin considerar la hora*

En el Listado 4-131, antes de comparar las fechas, anulamos la hora, los minutos y los segundos con el método setHours(). Ahora, las fechas tienen la hora configurada en 0 y el operador de comparación solo tiene que comprobar si las fechas son iguales o no. En este caso, las fechas son iguales y el mensaje "Las fechas son Iguales" se muestra en pantalla.

IMPORTANTE: además de los métodos estudiados en este capítulo, los objetos **`Date`** también incluyen métodos que consideran la localización (la ubicación física del usuario y su idioma). Debería considerar implementar estos métodos cuando desarrolle un sitio web o aplicación en español u otros idiomas diferentes del inglés, o cuando tenga que considerar variaciones horarias, como los desplazamientos introducidos durante los cambios de estación. MomentJS es una librería JavaScript que los desarrolladores usan a menudo para simplificar su trabajo y está disponible en www.momentjs.com. Estudiaremos librerías externas y cómo implementarlas más adelante en este capítulo.

4.5. Objeto Math

Algunos objetos en JavaScript no tienen la función de almacenar valores, sino la de proveer propiedades y métodos para procesar valores de otros objetos. Este es el caso del objeto Math. Este objeto no incluye un constructor para crear más objetos, pero define varias propiedades y métodos a los que podemos acceder desde su definición para obtener los valores de constantes matemáticas y realizar operaciones aritméticas. Los siguientes son los más usados.

PI—Esta propiedad devuelve el valor de PI.

E—Esta propiedad devuelve la constante Euler.

LN10—Esta propiedad devuelve el logaritmo natural de 10. El objeto también incluye las propiedades **`LN2`** (algoritmo natural de 2), **`LOG2E`** (algoritmo base 2 de E) y **`LOG10E`** (algoritmo base 10 de E).

SQRT2—Esta propiedad devuelve la raíz cuadrada de 2. El objeto también incluye la propiedad **`SQRT1_2`** (la raíz cuadrada de 1/2).

Ceil(valor**)**—Este método redondea un valor hacia arriba al siguiente entero y devuelveel resultado.

Floor(valor**)**—Este método redondea un valor hacia abajo al siguiente entero y devuelve el resultado.

Round(valor**)**—Este método redondea un valor al entero más cercano y devuelve el resultado.

Trunc(valor**)**—Este método elimina los dígitos de la fracción y devuelve un entero. **abs(**valor**)**—Este método devuelve el valor absoluto de un número (invierte valores negativos para obtener un número positivo).

Min(valor**)**—Este método devuelve el número más pequeño de una lista de valoresseparados por comas.

Max(valores**)**—Este método devuelve el número más grande de una lista de valores separados por comas.

Random()—Este método devuelve un número al azar en un rango entre 0 y 1.

Pow(base, exponente**)**—Este método devuelve el resultado de elevar la base a la potencia del exponente.

Exp(exponente**)**—Este método devuelve el resultado de elevar E a la potencia del exponente. El objeto también incluye el método **`expm1()`**, que devuelve el mismo resultado menos 1.

Sqrt(valor**)**—Este método devuelve la raíz cuadrada de un valor.

Log10(valor**)**—Este método devuelve el logaritmo base 10 de un valor. El objeto también incluye los métodos **`log()`** (devuelve el logaritmo base E), **`log2()`** (devuelve el logaritmobase 2) y **`log1p()`** (devuelve el logaritmo base E de 1 más un número).

Sin(valor**)**—Este método devuelve el seno de un número. El objeto también incluye los métodos **`asin()`** (devuelve el arcoseno de un número), **`sinh()`** (devuelve el seno hiperbólico de un número) y **`asinh()`** (devuelve el arcoseno hiperbólico de un número).

Cos(valor**)**—Este método devuelve el coseno de un número. El objeto también incluye los métodos **`acos()`** (devuelve el arcocoseno de un número), **`cosh()`** (devuelve el coseno hiperbólico de un número), y **`acosh()`** (devuelve el arcocoseno hiperbólico de un número).

Tan(valor**)**—Este método devuelve la tangente de un número. El objeto también incluye los métodos **`atan()`** (devuelve la arcotangente de un número), **`atan2()`** (devuelve la arcotangente o el cociente de dos números), **`tanh()`** (devuelve la tangente hiperbólica de un número), y **`atanh()`** (devuelve la arcotangente hiperbólica de un número).

Como no se crea ninguna instancia de este objeto, debemos leer las propiedades y llamar a los métodos desde el objeto mismo (por ejemplo, Math.PI). Aparte de esto, la aplicación de estos métodos y valores es sencilla, tal como muestra el siguiente ejemplo.

```
var cuadrado = Math.sqrt(4);  // 2
var elevado = Math.pow(2, 2);  // 4
var 301áximo = Math.max(cuadrado, elevado);

alert("El valor más grande es " + 301áximo);  // "El valor más grande
es 4"
```

Listado 4-132: *Ejecutando operaciones aritméticas con el objeto* `Math`

El código del Listado 4-132 obtiene la raíz cuadrada de 4, calcula 2 a la potencia de 2 y luego compara los resultados para obtener el mayor valor con el método max(). El siguiente ejemplo demuestra cómo calcular un número al azar.

```
var numeroalazar = Math.random() * (11 – 1) + 1;
var valor = Math.floor(numeroalazar);
alert("El número es: " + valor);
```

Listado 4-133: *Obteniendo un valor al azar*

El método random() devuelve un número entre 0 y 1. Si queremos obtener un número dentro de un rango diferente de valores, tenemos que multiplicar el valor por la fórmula **`(max – min) + min`**, donde **`min`** y **`max`** son los valores mínimos y máximos que queremos incluir. En nuestro ejemplo, queremos obtener un número al azar entre 1 y 10, pero hemos tenido que definirlo como 1 y 11 debido a que el valor máximo no se incluye en el rango. Otro tema que debemos considerar es que el número que devuelve el método **`random()`** es un valor decimal. Si queremos obtener un número entero, tenemos que redondearlo con el método **`floor()`**.

4.6. Objeto Window

Cada vez que abrimos el navegador o iniciamos una nueva pestaña, se crea un objeto global llamado Window para referenciar la ventana del navegador y proveer algunas propiedades y métodos esenciales. El objeto se almacena en una propiedad del objeto global de JavaScript llamada window. A través de esta propiedad podemos conectarnos con el navegador y el documento desde nuestro código.

El objeto Window a su vez incluye otros objetos con los que provee información adicional relacionada con la ventana y el documento. Las siguientes son algunas de las propiedades disponibles para acceder a estos objetos.

Location—Esta propiedad contiene un objeto **`Location`** con información acerca del origen del documento. También se puede usar como una propiedad para declarar o devolver la URL del documento (por ejemplo, **`window.location = "http://www.formasterminds.com"`**).

history—Esta propiedad contiene un objeto **`History`** con propiedades y métodos para manipular el historial de navegación. Estudiaremos este objeto en el Capítulo 19.

Navigator—Esta propiedad contiene un objeto **`Navigator`** con información acerca de la aplicación y el dispositivo. Estudiaremos algunas de las propiedades de este objeto, como **`geolocation`** (usada para detectar la ubicación del usuario) en próximos capítulos.

Document—Esta propiedad contiene un objeto **`Document`**, que provee acceso a los objetos que representan los elementos HTML en el documento.

Además de estos valiosos objetos, el objeto Window también ofrece sus propias propiedades y métodos. Los siguientes son los que más se usan:

innerWidth—Esta propiedad devuelve el ancho de la ventana en píxeles.

innerHeight—Esta propiedad devuelve la altura de la ventana en píxeles.

scrollX—Esta propiedad devuelve el número de píxeles en los que el documento se ha desplazado horizontalmente.

scrollY—Esta propiedad devuelve el número de píxeles en los que el documento se ha desplazado verticalmente.

Alert(valor**)**—Este método muestra una ventana emergente en la pantalla que muestra el valor entre paréntesis.

Confirm(mensaje**)**—Este método es similar a **`alert()`**, pero ofrece dos botones, Aceptar y Cancelar, para que el usuario elija qué hacer. El método devuelve **`true`** o **`false`**, según a la respuesta del usuario.

Prompt(mensaje**)**—Este método muestra una ventana emergente con un campo de entrada para permitir al usuario introducir un valor. El método devuelve el valor que inserta el usuario.

setTimeout(función, milisegundos**)**—Este método ejecuta la función especificada en el primer atributo cuando haya pasado el tiempo especificado por el segundo atributo. El objeto **`Window`** también ofrece el método **`clearTimeout()`** para cancelar este proceso.

setInterval(función, milisegundos**)**—Este método es similar a **`setTimeout()`**, pero llama a la función constantemente. El objeto **`Window`** también ofrece el método **`clearInterval()`** para cancelar el proceso.

Open(URL, ventana, parámetros**)**—Este método abre un documento en una nueva ventana. El atributo **URL** es la URL del documento que queremos abrir, el atributo **ventana** es el nombre de la ventana donde queremos mostrar el documento (si el nombre no se especifica o la ventana no existe, el documento se abre en una nueva ventana), y el atributo **parámetros** es una lista de parámetros de configuración separados por comas que configuran las características de la ventana (por ejemplo, "resizable=no,scrollbars=no"). El objeto **`Window`** también ofrece el método **`close()`** para cerrar una ventana abierta con este método.

El objeto Window controla aspectos de la ventana, su contenido, y los datos asociados a la misma, como la ubicación del documento actual, el tamaño, el desplazamiento, etc. Por ejemplo, podemos cargar un nuevo documento cambiando el valor de la ubicación.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function realizar() {
      window.location = "http://www.formasterminds.com";
    }
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <button type="button" onclick="realizar()">Presione Aquí</button>
  </section>
</body>
</html>
```

Listado 4-134: *Definiendo una nueva ubicación*

En el Listado 4-134 declaramos una función llamada realizar() que asigna una nueva URL a la propiedad location del objeto Window. Luego se asigna una llamada a la función al atributo onclick del elemento <button> para ejecutar la función cuando se pulsa el botón.

La propiedad location contiene un objeto Location con sus propias propiedades y métodos, pero también podemos asignar una cadena de caracteres con la URL directamente a la propiedad para definir una nueva ubicación para el contenido de la ventana. Una vez que el valor se asigna a la propiedad, el navegador carga el documento en esa URL y lo muestra en la pantalla.

Ejercicio 9:cree un nuevo archivo HTML con el Listado 4-134 y abra el documento en su navegador. Debería ver un título y un botón. Pulse el botón. El navegador debería cargar el sitio web **www.formasterminds.com**.

Además de asignar una nueva URL a la propiedad location, también podemos manipular la ubicación desde los métodos provistos por el objeto Location.

Assign(URL**)**—Este método le pide al navegador que cargue el documento en la ubicación especificada por el atributo **URL**.

Replace(URL**)**—Este método le pide al navegador que reemplace el documento actual con el documento en la ubicación indicada por el atributo **URL**. Difiere del método **`assign()`** en que no agrega la URL al historial del navegador.

Reload(valor**)**—Este método le pide al navegador que actualice el documento actual. Acepta un valor booleano que determina si el recurso se tiene que descargar desde el servidor o se puede cargar desde el caché del navegador (**`true`** o **`false`**).

El siguiente ejemplo actualiza la página cuando el usuario pulsa un botón. Esta vez no mencionamos la propiedad window. El objeto Window es un objeto global y, por lo tanto, el intérprete infiere que las propiedades y los métodos pertenecen a este objeto. Esta es la razón por la que en anteriores ejemplos nunca hemos llamado al método alert() con la instrucción window.alert().

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function realizar() {
      location.reload();
    }
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <button type="button" onclick="realizar()">Presione Aquí</button>
  </section>
</body>
</html>
```

Listado 4-135: *Actualizando la página*

El objeto Window también ofrece el método open() para cargar nuevo contenido. En el siguiente ejemplo, el sitio web www.formasterminds.com se abre en una nueva ventana o pestaña.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function realizar() {
      open("http://www.formasterminds.com");
    }
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <button type="button" onclick="realizar()">Presione Aquí</button>
  </section>
</body>
</html>
```

Listado 4-136: *Abriendo una nueva ventana*

Otros métodos del objeto Window son setTimeout() y setInterval(). Estos métodos ejecutan una instrucción después de un cierto período de tiempo. El método setTimeout() ejecuta la instrucción una vez, y el método setInterval() ejecuta la instrucción de forma repetida hasta que el proceso se cancela. Si en lugar de una instrucción queremos ejecutar varias, podemos especificar una referencia a una función. Cada vez que el tiempo finaliza, la función se ejecuta.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function realizar() {
      var hoy = new Date();
      var tiempo = hoy.toTimeString();
      alert(tiempo);
    }
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <button type="button" onclick="setTimeout(realizar, 5000)">Presione Aquí</button>
  </section>
</body>
</html>
```

Listado 4-137: *Usando un temporizador para ejecutar funciones*

Estos métodos aceptan valores en milisegundos. Un milisegundo son 1000 partes de un segundo, por lo que si queremos especificar el tiempo en segundos, tenemos que multiplicar el valor por 1000. En nuestro ejemplo, queremos que la función realizar() se ejecute cada 5 segundos y, por lo tanto, declaramos el valor 5000 como el tiempo que el código debe esperar para llamar a la función.

IMPORTANTE: la función se declara sin los paréntesis. Cuando queremos llamar a una función, tenemos que declarar los paréntesis después del nombre, pero cuando queremos referenciar una función, debemos omitir los paréntesis. Como en esta oportunidad queremos asignar una referencia a la función y no el resultado de su ejecución, declaramos el nombre sin paréntesis.

Si lo que necesitamos es ejecutar la función una y otra vez después de un período de tiempo, tenemos que usar el método setInterval(). Este método trabaja exactamente como setTimeout() pero sigue funcionando hasta que le pedimos que se detenga con el método clearInterval(). Para identificar al método que queremos que se cancele, tenemos que almacenar la referencia que devuelve el método setInterval() en una variable y usar esa variable para referenciar el método más adelante, tal como hacemos en el siguiente ejemplo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    var segundos = 0;
    var contador = setInterval(realizar, 1000);
    function realizar() {
      segundos++;
    }
    function cancelar() {
      clearInterval(contador);
      alert("Total: " + segundos + " segundos");
    }
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <button type="button" onclick="cancelar()">Presione Aquí</button>
  </section>
</body>
</html>
```

Listado 4-138: *Cancelando un temporizador*

El código del Listado 4-138 incluye dos funciones. La función realizar() se ejecuta cada 1 segundo mediante el método setInterval() y la función cancelar() se ejecuta cuando el usuario pulsa el botón. El propósito de este código es incrementar el valor de una variable llamada segundos cada segundo hasta que el usuario decide cancelar el proceso y ver el total acumulado hasta el momento.

Ejercicio 10:reemplace el documento en su archivo HTML con el código del Listado 4-138 y abra el nuevo documento en su navegador. Espere un momento y pulse el botón. Debería ver una ventana emergente con el número de segundos que han pasado hasta el momento.

IMPORTANTE: los métodos **`setTimeout()`** y **`setInterval()`** son necesarios en la construcción de pequeñas aplicaciones y animaciones. Estudiaremos estos métodos en situaciones más prácticas en próximos capítulos.

4.7. Objeto Document

Como ya hemos mencionado, casi todo en JavaScript se define como un objeto, y esto incluye los elementos en el documento. Cuando se carga un documento HTML, el navegador crea una estructura interna para procesarlo. La estructura se llama DOM (Document Object Model) y está compuesta por múltiples objetos de tipo Element (u otros tipos más específicos que heredan de Element), que representan cada elemento en el documento.

Los objetos Element mantienen una conexión permanente con los elementos que representan. Cuando se modifica un objeto, su elemento también se modifica y el resultado se muestra en pantalla. Para ofrecer acceso a estos objetos y permitirnos alterar sus propiedades desde nuestro código JavaScript, los objetos se almacenan en un objeto llamado Document que se asigna a la propiedad document del objeto Window.

Entre otras alternativas, el objeto Document incluye las siguientes propiedades para ofrecer acceso rápido a los objetos Element que representan los elementos más comunes del documento.

forms—Esta propiedad devuelve un array con referencias a todos los objetos **`Element`** que representan los elementos **`<form>`** en el documento.

images—Esta propiedad devuelve un array con referencias a todos los objetos **`Element`** que representan los elementos **`<img>`** en el documento.

links—Esta propiedad devuelve un array con referencias a todos los objetos **`Element`** que representan los elementos **`<a>`** en el documento.

Estas propiedades devuelven un array de objetos que referencian todos los elementos de un tipo particular, pero el objeto Document también incluye los siguientes métodos para acceder a objetos individuales u obtener listas de objetos a partir de otros parámetros.

getElementById(id**)**—Este método devuelve una referencia al objeto **`Element`** que representa el elemento identificado con el valor especificado por el atributo (el valor asignado al atributo **`id`**).

getElementsByClassName(clase**)**—Este método devuelve un array con referencias a los objetos **`Element`** que representan los elementos identificados con la clase especificada por el atributo (el valor asignado al atributo **`clase`**).

getElementsByName(nombre**)**—Este método devuelve un array con referencias a los objetos **`Element`** que representan los elementos identificados con el nombre especificado por el atributo (el valor asignado al atributo **`name`**).

getElementsByTagName(tipo**)**—Este método devuelve un array con referencias a los objetos **`Element`** que representan el tipo de elementos especificados por el atributo. El atributo es el nombre que identifica a cada tipo de elemento, como **`h1`**, **`p`**, **`img`**, **`div`**, etc.

querySelector(selector**)**—Este método devuelve una referencia al objeto **`Element`** que representa el elemento que coincide con el selector especificado por el atributo. El método devuelve el primer elemento en el documento que coincide con el selector CSS (ver Capítulo 3 para consultar cómo construir estos selectores).

querySelectorAll(selectores**)**—Este método devuelve un array con referencias a los objetos **`Element`** que representan los elementos que coinciden con los selectores especificados por el atributo. Se pueden declarar uno o más selectores separados por comas.

Accediendo a los objetos Element en el DOM, podemos leer y modificar los elementos en el documento, pero antes de hacerlo, debemos considerar que el navegador lee el documento de forma secuencial y no podemos referenciar un elemento que aún no se ha creado. La mejor solución a este problema es ejecutar el código JavaScript solo cuando ocurre el evento load. Ya hemos estudiado este evento al comienzo de este capítulo. Los navegadores disparan el evento después de que se ha cargado el documento y todos los objetos en el DOM se han creado y son accesibles. El siguiente ejemplo incluye el atributo onload en el elemento <body> para poder acceder a un elemento <p> en la cabecera del documento desde el código JavaScript.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function iniciar() {
      var elemento = document.getElementById("subtitulo");
    }
  </script>
</head>
<body onload="iniciar()">
  <section>
    <h1>Website</h1>
    <p id="subtitulo">El mejor sitio web!</p>
  </section>
</body>
</html>
```

Listado 4-139: *Obteniendo una referencia al objeto* `Element` *que representa un elemento*

El código del Listado 4-139 no realiza ninguna acción; lo único que hace es obtener una referencia al objeto Element que representa el elemento <p> y la almacena en la variable elemento tan pronto como se carga el documento. Para hacer algo con el elemento, tenemos que trabajar con las propiedades del objeto.

Cada objeto Element obtiene automáticamente propiedades que referencian cada atributo del elemento que representan. Leyendo estas propiedades podemos obtener o modificar los valores de los atributos correspondientes. Por ejemplo, si leemos la propiedad id en el objeto almacenado en la variable elemento, obtenemos la cadena de caracteres "subtitulo".

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function iniciar() {
      var elemento = document.getElementById("subtitulo");
      alert("El id es: " + elemento.id);  // "El id es: subtitulo"
    }
  </script>
</head>
<body onload="iniciar()">
  <section>
    <h1>Website</h1>
    <p id="subtitulo">El mejor sitio web!</p>
  </section>
</body>
</html>
```

Listado 4-140: *Leyendo los atributos de los elementos desde JavaScript*

En estos ejemplos hemos accedido al elemento con el método getElementById() porque el elemento <p> de nuestro documento tiene un atributo id, pero no siempre podemos contar con esto. Si el atributo id no está presente o queremos obtener una lista de elementos que comparten similares características, podemos aprovechar el resto de los métodos provistos por el objeto Document. Por ejemplo, podemos obtener una lista de todos los elementos <p> del documento con el método getElementsByTagName().

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function iniciar() {
      var lista = document.getElementsByTagName("p");
      for (var f = 0; f < lista.length; f++) {
        var elemento  =  lista[f];
        alert("El id es: " + elemento.id);
      }
    }
  </script>
</head>
<body onload="iniciar()">
  <section>
    <h1>Sitio Web</h1>
    <p id="subtitulo">El mejor sitio web!</p>
  </section>
</body>
</html>
```

Listado 4-141: *Accediendo a elementos por el nombre*

El método getElementsByTagName() devuelve un array con referencias a todos los elementos cuyos nombres son iguales al valor provisto entre paréntesis. En el ejemplo del Listado 4-141, el atributo se ha definido con el texto "p", por lo que el método devuelve una lista de todos los elementos <p> en el documento. Después de obtener las referencias, accedemos a cada elemento con un bucle for y mostramos el valor de sus atributos id en la pantalla.

Si conocemos la posición del elemento que queremos acceder, podemos especificar su índice. En nuestro ejemplo, solo tenemos un único elemento <p>, por lo tanto la referencia a este elemento se encontrará en la posición 0 del array.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function iniciar() {
      var lista = document.getElementsByTagName("p");
      var elemento = lista[0];
      alert("El id es: " + elemento.id);
    }
  </script>
</head>
<body onload="iniciar()">
  <section>
    <h1>Sitio Web</h1>
    <p id="subtitulo">El mejor sitio web!</p>
  </section>
</body>
</html>
```

Listado 4-142: *Accediendo a un elemento por medio de su nombre*

Otro método para encontrar elementos es querySelector(). Este método busca un elemento usando un selector CSS. La ventaja es que podemos explotar toda la capacidad de los selectores CSS para encontrar el elemento correcto. En el siguiente ejemplo, usamos el método querySelector() para encontrar elementos <p> que son hijos directos de un elemento <section>.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function iniciar() {
      var elemento = document.querySelector("section > p");
      alert("El id es: " + elemento.id);
    }
  </script>
</head>
<body onload="iniciar()">
  <section>
    <h1>Sitio Web</h1>
    <p id="subtitulo">El mejor sitio web!</p>
  </section>
</body>
</html>
```

Listado 4-143: *Buscando un elemento con el método* `querySelector()`

Lo básico: el método **`querySelector()`** devuelve solo la referencia al primer elemento encontrado. Si queremos obtener una lista de todos los elementos que coinciden con el selector, tenemos que usar el método **`querySelectorAll()`**.

Los métodos que hemos estudiado buscan elementos en todo el documento, pero podemos reducir la búsqueda buscando solo en el interior de un elemento. Con este propósito, los objetos Element también incluyen sus propias versiones de métodos como getElementsByTagName() y querySelector(). Por ejemplo, podemos buscar elementos <p> dentro de elementos <section> con una identificación específica.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function iniciar() {
      var elemprincipal = document.getElementById("seccionprincipal");
      var lista = elemprincipal.getElementsByTagName("p");
      var elemento = lista[0];
      alert("El id es: " + elemento.id);
    }
  </script>
</head>
<body onload="iniciar()">
  <section id="seccionprincipal">
    <h1>Sitio Web</h1>
    <p id="subtitulo">El mejor sitio web!</p>
  </section>
</body>
</html>
```

Listado 4-144: *Buscando un elemento dentro de otro elemento*

El código del Listado 4-144 obtiene una referencia al elemento identificado con el nombre "seccionprincipal" y luego llama al método getElementsByTagName() para encontrar los elementos <p> dentro de ese elemento. Debido a que solo tenemos un elemento <p> dentro del elemento <section>, leemos la referencia en el índice 0 y mostramos el valor de su atributo id en la pantalla.

4.8. Objetos Element

Obtener una referencia para acceder a un elemento y leer sus atributos puede ser útil en algunas circunstancias, pero lo que convierte a JavaScript en un lenguaje dinámico es la posibilidad de modificar esos elementos y el documento. Con este propósito, los objetos Element contienen propiedades para manipular y definir los estilos de los elementos y sus contenidos. Una de estas propiedades es style, el cual contiene un objeto llamado Styles que a su vez incluye propiedades para modificar los estilos de los elementos.

Los nombres de los estilos en JavaScript no son los mismos que en CSS. No hubo consenso a este respecto y, a pesar de que podemos asignar los mismos valores a las propiedades, tenemos que aprender sus nombre en JavaScript. Las siguientes son las propiedades que más se usan.

color—Esta propiedad declara el color del contenido del elemento.

background—Esta propiedad declara los estilos del fondo del elemento. También podemos trabajar con cada estilo de forma independiente usando las propiedades asociadas **backgroundColor**, **backgroundImage**, **backgroundRepeat**, **backgroundPosition** y **backgroundAttachment**.

border—Esta propiedad declara los estilos del borde del elemento. Podemos modificar cada estilo de forma independiente con las propiedades asociadas **borderColor**, **borderStyle**, y **borderWidth**, o modificar cada borde individualmente usando las propiedades asociadas **borderTop** (**borderTopColor**, **borderTopStyle**, y **borderTopWidth**), **borderBottom** (**borderBottomColor**, **borderBottomStyle**, y **borderBottomWidth**), **borderLeft** (**borderLeftColor**, **borderLeftStyle**, y **borderLeftWidth**), y **borderRight** (**borderRightColor**, **borderRightStyle**, y **borderRightWidth**).

margin—Esta propiedad declara el margen del elemento. También podemos usar las propiedades asociadas **marginBottom**, **marginLeft**, **marginRight**, y **marginTop**.

padding—Esta propiedad declara el relleno del elemento. También podemos usar las propiedades asociadas **paddingBottom**, **paddingLeft**, **paddingRight**, y **paddingTop**.

width—Esta propiedad declara el ancho del elemento. Existen dos propiedades asociadas para declarar el ancho máximo y mínimo de un elemento: **maxWidth** y **minWidth**.

height—Esta propiedad declara la altura del elemento. Existen dos propiedades asociadas para declarar la altura máxima y mínima de un elemento: **maxHeight** y **minHeight**.

visibility—Esta propiedad determina si el elemento es visible o no.

display—Esta propiedad define el tipo de caja usado para presentar el elemento.

position—Esta propiedad define el tipo de posicionamiento usado para posicionar el elemento.

top—Esta propiedad especifica la distancia entre el margen superior del elemento y el margen superior de su contenedor.

bottom—Esta propiedad especifica la distancia entre el margen inferior del elemento y el margen inferior de su contenedor.

left—Esta propiedad especifica la distancia entre el margen izquierdo del elemento y el margen izquierdo de su contenedor.

right—Esta propiedad especifica la distancia entre el margen derecho del elemento y el margen derecho de su contenedor.

cssFloat—Esta propiedad permite al elemento flotar hacia un lado o el otro.

clear—Esta propiedad recupera el flujo normal del documento, impidiendo que el elemento siga flotando hacia la izquierda, la derecha, o ambos lados.

overflow—Esta propiedad especifica cómo se va a mostrar el contenido que excede los límites de la caja de su contenedor.

zIndex—Esta propiedad define un índice que determina la posición del elemento en el eje z.

font—Esta propiedad declara los estilos de la fuente. También podemos declarar los estilos individuales usando las propiedades asociadas **`fontFamily`**, **`fontSize`**, **`fontStyle`**, **`fontVariant`** y **`fontWeight`**.

textAlign—Esta propiedad alinea el texto dentro del elemento.

verticalAlign—Esta propiedad alinea elementos Inline verticalmente.

textDecoration—Esta propiedad resalta el texto con una línea. También podemos declarar los estilos de forma individual asignando los valores **`true`** o **`false`** a las propiedades **`textDecorationBlink`**, **`textDecorationLineThrough`**, **`textDecorationNone`**, **`textDecorationOverline`**, y **`textDecorationUnderline`**.

Modificar los valores de estas propiedades es sencillo. Debemos obtener una referencia al objeto Element que representa el elemento que queremos modificar, como lo hemos hecho en ejemplos anteriores, y luego asignar un nuevo valor a la propiedad del objeto Styles que queremos cambiar. La única diferencia con CSS, además de los nombres de las propiedades, es que los valores se tienen que asignar entre comillas.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function iniciar() {
      var elemento = document.getElementById("subtitulo");
      elemento.style.width = "300px";
      elemento.style.border = "1px solid #FF0000";
      elemento.style.padding = "20px";
    }
  </script>
</head>
<body onload="iniciar()">
  <section>
    <h1>Sitio Web</h1>
    <p id="subtitulo">El mejor sitio web!</p>
  </section>
</body>
</html>
```

Listado 4-145: *Asignando nuevos estilos al documento*

El código del Listado 4-145 asigna un ancho de 300 píxeles, un borde rojo de 1 píxel, y un relleno de 20 píxeles al elemento <p> del documento. El resultado es el que se muestra en la Figura 4-4.

Figura 4-4: *Estilos asignados desde JavaScript*

Las propiedades del objeto Styles son independientes de los estilos CSS asignados al documento. Si intentamos leer una de estas propiedades pero aún no le hemos asignado ningún valor desde JavaScript, el valor que devuelve será una cadena de caracteres vacía. Para proveer información acerca del elemento, los objetos Element incluyen propiedades adicionales. Las siguientes son las que más se usan.

clientWidth—Esta propiedad devuelve el ancho del elemento, incluido el relleno.

clientHeight—Esta propiedad devuelve la altura del elemento, incluido el relleno.

offsetTop—Esta propiedad devuelve el número de píxeles que se ha desplazado el elemento desde la parte superior de su contenedor.

offsetLeft—Esta propiedad devuelve el número de píxeles que se ha desplazado el elemento desde el lado izquierdo de su contenedor.

offsetWidth—Esta propiedad devuelve el ancho del elemento, incluidos el relleno y el borde.

offsetHeight—Esta propiedad devuelve la altura del elemento, incluidos el relleno y el borde.

scrollTop—Esta propiedad devuelve el número de píxeles en los que el contenido del elemento se ha desplazado hacia arriba.

scrollLeft—Esta propiedad devuelve el número de píxeles en los que el contenido del elemento se ha desplazado hacia la izquierda.

scrollWidth—Esta propiedad devuelve el ancho del contenido del elemento.

scrollHeight—Esta propiedad devuelve la altura del contenido del elemento.

Estas son propiedades de solo lectura, pero podemos obtener el valor que necesitamos leyendo estas propiedades y después usar las propiedades del objeto Styles para asignarle uno nuevo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <style>
    #subtitulo {
      width: 300px;
      padding: 20px;
      border: 1px solid #FF0000;
    }
  </style>
  <script>
    function iniciar() {
      var elemento = document.getElementById("subtitulo");
      var ancho = elemento.clientWidth;
      ancho = ancho + 100;
      elemento.style.width = ancho + "px";
    }
  </script>
</head>
<body onload="iniciar()">
  <section>
    <h1>Sitio Web</h1>
    <p id="subtitulo">El mejor sitio web!</p>
  </section>
</body>
</html>
```

Listado 4-146: *Leyendo estilos CSS desde JavaScript*

En este ejemplo, asignamos los estilos al elemento <p> desde CSS y luego modificamos su ancho desde JavaScript. El ancho actual se toma de la propiedad clientWidth, pero debido a que esta propiedad es de solo lectura, se tiene que asignar el nuevo valor a la propiedad width del objeto Styles (el valor asignado a la propiedad debe ser una cadena de caracteres con las unidades "px" al final). Una vez se ejecuta el código, el elemento <p> tiene un ancho de 400 píxeles.

Figura 4-5: *Estilos modificados desde JavaScript*

No es común que modifiquemos los estilos de un elemento uno por uno, como hemos hecho en estos ejemplos. Normalmente, los estilos se asignan a los elementos desde grupos de propiedades CSS a través del atributo class. Como hemos explicado en el Capítulo 3, estas reglas se llaman clases. Las clases se definen de forma permanente en hojas de estilo CSS, pero los objetos Element incluyen las siguientes propiedades para asignar una clase diferente a un elemento y, por lo tanto, modificar sus estilos todos a la vez.

className—Esta propiedad declara o devuelve el valor del atributo `class`.

classList—Esta propiedad devuelve un array con la lista de las clases asignadas al elemento.

El array que devuelve la propiedad classList es de tipo DOMTokenList, que incluye los siguientes métodos para modificar las clases de la lista.

add(clase**)**—Este método agrega una clase al elemento.

remove(class**)**—Este método agrega una clase del elemento.

toggle(clase**)**—Este método agrega o elimina una clase dependiendo del estado actual. Si la clase ya se ha asignado al elemento, la elimina, y en caso contrario la agrega.

contains(clase**)**—Este método detecta si se ha asignado la clase al elemento o no, y devuelve `true` o `false` respectivamente.

La forma más fácil de reemplazar la clase de un elemento es asignando un nuevo valor a la propiedad className.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <style>
    .supercolor {
      background: #0099EE;
    }
    .colornegro {
      background: #000000;
    }
  </style>
  <script>
    function cambiarcolor() {
      var elemento = document.getElementById("subtitulo");
      elemento.className = "colornegro";
    }
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <p id="subtitulo" class="supercolor" onclick="cambiarcolor()">El
mejor sitio web!</p>
  </section>
</body>
</html>
```

Listado 4-147: *Reemplazando la clase del elemento*

En el código del Listado 4-147 hemos declarado dos clases: supercolor y colornegro. Ambas definen el color de fondo del elemento. Por defecto, la clase supercolor se asigna al elemento <p>, lo que le otorga al elemento un fondo azul, pero cuando se ejectua la función cambiarcolor(), esta clase se reemplaza por la clase colornegro y el color negro se asigna al fondo (esta vez ejecutamos la función cuando el usuario hace clic en el elemento, no cuando el documento termina de cargarse).

Como hemos mencionado en el Capítulo 3, se pueden asignar varias clases a un mismo elemento. Cuando esto ocurre, en lugar de la propiedad className es mejor utilizar los métodos de la propiedad classList. El siguiente ejemplo implementa el método contains() para detectar si ya se ha asignado una clase a un elemento y la agrega o la elimina, dependiendo del estado actual.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <style>
    .supercolor {
      background: #000000;
    }
  </style>
  <script>
    function cambiarcolor() {
      var elemento = document.getElementById("subtitulo");
      if (elemento.classList.contains("supercolor")) {
        elemento.classList.remove("supercolor");
      } else {
        elemento.classList.add("supercolor");
      }
    }
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <p id="subtitulo" class="supercolor" onclick="cambiarcolor()">El
mejor sitio web!</p>
  </section>
</body>
</html>
```

Listado 4-148: *Activando y desactivando clases*

Con el código del Listado 4-148, cada vez que el usuario hace clic en el elemento <p>, su estilo se modifica, pasando de tener un fondo de color a no tener ningún fondo. Se puede obtener el mismo efecto con el método toggle(). Este método comprueba el estado del elemento y agrega la clase si no se ha asignado anteriormente, o la elimina en caso contrario.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <style>
    .supercolor {
      background: #000000;
    }
  </style>
  <script>
    function cambiarcolor() {
      var elemento = document.getElementById("subtitulo");
      elemento.classList.toggle("supercolor");
    }
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <p id="subtitulo" class="supercolor" onclick="cambiarcolor()">El
mejor sitio web!</p>
  </section>
</body>
</html>
```

Listado 4-149: *Activando y desactivando clases con el método* `toggle()`

El método toogle() simplifica nuestro trabajo. Ya no tenemos que controlar si la clase existe o no, el método lo hace por nosotros y agrega la clase o la elimina dependiendo del estado actual.

Ejercicio 11:cree un nuevo archivo HTML con el documento que quiere probar. Abra el documento en su navegador y haga clic en el área que ocupa el elemento **<p>**. Debería ver cómo el fondo del elemento cambia de color.

Además de los estilos de un elemento, también podemos modificar su contenido. Estas son algunas de las propiedades y métodos provistos por los objetos Element con este propósito.

innerHTML—Esta propiedad declara o devuelve el contenido de un elemento.

outerHTML—Esta propiedad declara o devuelve un elemento y su contenido. A diferencia de la propiedad **`innerHTML`**, esta propiedad no solo reemplaza el contenido, sino también el elemento.

insertAdjacentHTML(ubicación, contenido**)**—Este método inserta contenido en una ubicación determinada por el atributo **ubicación**. Los valores disponibles son **`beforebegin`** (antes del elemento), **`afterbegin`** (dentro del elemento, antes del primer elemento hijo), **`beforeend`** (dentro del elemento, después del último elemento hijo) y **`afterend`** (después del elemento).

La manera más sencilla de reemplazar el contenido de un elemento es con la propiedad innerHTML. Asignando un nuevo valor a esta propiedad, el contenido actual se reemplaza con el nuevo. El siguiente ejemplo reemplaza el contenido del elemento <p> con el texto "Este es mi sitio web" cuando hacemos clic en él.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function cambiarcontenido() {
      var elemento = document.getElementById("subtitulo");
      elemento.innerHTML = "Este es mi sitio web";  }
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <p id="subtitulo" onclick="cambiarcontenido()">El mejor sitio web!</p>
  </section>
</body>
</html>
```

Listado 4-150: *Asignando contenido a un elemento*

La propiedad innerHTML no solo se utiliza para asignar nuevo contenido, sino también para leer y procesar el contenido actual. El siguiente ejemplo lee el contenido de un elemento, le agrega un texto al final y asigna el resultado de vuelta al mismo elemento.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function cambiarcontenido() {
      var elemento = document.getElementById("subtitulo");
      var texto = elemento.innerHTML + " Somos los mejores!";
      elemento.innerHTML = texto;
    }
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <p id="subtitulo" onclick="cambiarcontenido()">El mejor sitio
web!</p>
  </section>
</body>
</html>
```

Listado 4-151: *Modificando el contenido de un elemento*

Además de texto, la propiedad innerHTML también puede procesar código HTML. Cuando el código HTML se asigna a esta propiedad, se interpreta y el resultado se muestra en pantalla.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function agregarelemento() {
      var elemento = document.querySelector("section");
      elemento.innerHTML = "<p>Este es un texto nuevo</p>";
    }
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <button type="button" onclick="agregarelemento()">Agregar
Contenido</button>
  </section>
</body>
</html>
```

Listado 4-152: *Insertando código HTML en el documento*

El código del Listado 4-152 obtiene una referencia al primer elemento <section> en el documento y reemplaza su contenido con un elemento <p>. A partir de ese momento, el usuario solo verá el elemento <p> en la pantalla.

Si no queremos reemplazar todo el contenido de un elemento, sino agregar más contenido, podemos usar el método insertAdjacentHTML(). Este método puede agregar contenido antes o después del contenido actual y también fuera del elemento, dependiendo del valor asignado al primer atributo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function agregarelemento() {
      var elemento = document.querySelector("section");
      elemento.insertAdjacentHTML("beforeend", "<p>Este es un texto
nuevo</p>");
    }
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <button type="button" onclick="agregarelemento()">Agregar
Contenido</button>
  </section>
</body>
</html>
```

Listado 4-153: *Agregando contenido HTML dentro de un elemento*

El método insertAdjacentHTML() agrega contenido al documento, pero sin que afecte al contenido existente. Cuando pulsamos el botón en el documento del Listado 4-153, el código JavaScript agrega un elemento <p> debajo del elemento <button> (al final del contenido del elemento <section>). El resultado se muestra en la Figura 4-6.

Figura 4-6: *Contenido agregado a un elemento*

4.9. Creando objetos Element

Cuando se agrega código HTML al documento a través de propiedades y métodos como innerHTML o insertAdjacentHTML(), el navegador analiza el documento y genera los objetos Element necesarios para representar los nuevos elementos. Aunque es normal utilizar este procedimiento para modificar la estructura de un documento, el objeto Document incluye métodos para trabajar directamente con los objetos Element.

createElement(nombre**)**—Este método crea un nuevo objeto **`Element`** del tipo especificado por el atributo **nombre**.

appendChild(elemento**)**—Este método inserta el elemento representado por un objeto **`Element`** como hijo de un elemento existente en el documento.

removeChild(elemento**)**—Este método elimina un elemento hijo de otro elemento. El atributo debe ser una referencia del hijo a eliminarse.

Si nuestra intención es crear un nuevo objeto Element para agregar un elemento al documento, primero tenemos que crear el objeto con el método createElement() y luego usar este objeto para agregar el elemento al documento con el método appendChild().

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function agregarelemento() {
      var elemento = document.querySelector("section");
      var elementonuevo = document.createElement("p");
      elemento.appendChild(elementonuevo);
    }
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <button type="button" onclick="agregarelemento()">Agregar
Elemento</button>
  </section>
</body>
</html>
```

Listado 4-154: *Creando objetos* `Element`

El código del Listado 4-154 agrega un elemento <p> al final del elemento <section>, pero el elemento no tiene ningún contenido, por lo que no produce ningún cambio en la pantalla. Si queremos definir el contenido del elemento, podemos asignar un nuevo valor a su propiedad innerHTML. Los objetos Element que devuelve el método createElement() son los mismos que los creados por el navegador para representar el documento y, por lo tanto, podemos modificar sus propiedades para asignar nuevos estilos o definir sus contenidos. El siguiente código asigna contenido a un objeto Element antes de agregar el elemento al documento.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function agregarelemento() {
      var elemento = document.querySelector("section");
      var elementonuevo = document.createElement("p");
      elementonuevo.innerHTML = "Este es un elemento nuevo";
      elemento.appendChild(elementonuevo);
    }
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <button type="button" onclick="agregarelemento()">Agregar
Elemento</button>
  </section>
</body>
</html>
```

Listado 4-155: *Agregando contenido a un objeto* `Element`

Figura 4-7: *Elemento agregado al documento*

Lo básico: no hay mucha diferencia entre agregar los elementos con la propiedad **`innerHTML`** o estos métodos, pero el método **`createElement()`** resulta útil cuando trabajamos con aquellas API que requieren objetos **`Element`** para procesar información, como cuando tenemos que procesar una imagen o un vídeo que no se va a mostrar en pantalla, sino que se envía a un servidor o se almacena en el disco duro del usuario. Aprenderemos más acerca de las API en este capítulo y estudiaremos aplicaciones prácticas del método **`createElement()`** más adelante.

Capítulo 5
Eventos

5.1. Eventos

Como ya hemos visto, HTML provee atributos para ejecutar código JavaScript cuando ocurre un evento. En ejemplos recientes hemos implementado el atributo onload para ejecutar una función cuando el navegador termina de cargar el documento y el atributo onclick que ejecuta código JavaScript cuando el usuario hace clic en un elemento. Lo que no hemos mencionado es que estos atributos, como cualquier otro atributo, se pueden configurar desde JavaScript. Esto se debe a que, como también hemos visto, los atributos de los elementos se convierten en propiedades de los objetos Element y, por lo tanto, podemos definir sus valores desde código JavaScript. Por ejemplo, si queremos responder al evento **`click`**, solo tenemos que definir la propiedad **`onclick`** del elemento.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function agregarevento() {
      var elemento = document.querySelector("section > button");
      elemento.onclick = mostrarmensaje;}
    function mostrarmensaje() {
      alert("Presionó el
      botón");}
    window.onload = agregarevento;
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <button type="button">Mostrar</button>
  </section>
</body>
</html>
```

Listado 4-156: *Definiendo atributos de eventos desde código JavaScript*

El documento del Listado 4-156 no incluye ningún atributo de eventos dentro de los elementos; todos se declaran en el código JavaScript. En este caso, definimos dos atributos: el atributo onload del objeto Window y el atributo onclick del elemento <button>. Cuando se carga el documento, el evento load se desencadena y se ejectua la función agregarevento(). En esta función obtenemos una referencia al elemento <button> y definimos su atributo onclick para ejecutar la función mostrarmensaje() cuando se pulsa el botón. Con esta información, el documento está listo para trabajar. Si el usuario pulsa el botón, se muestra un mensaje en la pantalla.

Lo básico: no hay ninguna diferencia entre declarar el atributo `onload` en el elemento `<body>` o en el objeto `Window`, pero debido a que siempre debemos separar el código JavaScript del documento HTML y desde el código es más fácil definir el atributo en el objeto `Window`, esta es la práctica recomendada.

5.2. El método addEventListener()

No se recomienda el uso de atributos de evento en elementos HTML porque es contrario al propósito principal de HTML5 que es el de proveer una tarea específica para cada uno de los lenguajes involucrados. HTML debe definir la estructura del documento, CSS su presentación y JavaScript su funcionalidad. Pero la definición de estos atributos desde el código JavaScript, como hemos hecho en el ejemplo anterior, tampoco se recomienda. Por estas razones, se han incluido nuevos métodos en el objeto `Window` para controlar y responder a eventos.

addEventListener(evento, listener, captura**)**—Este método prepara un elemento para responder a un evento. El primer atributo es el nombre del evento (sin el prefijo `on`), el segundo atributo es una referencia a la función que responderá al evento (llamada *listener*) y el tercer atributo es un valor booleano que determina si el evento va a ser capturado por el elemento o se propagará a otros elementos (generalmente se ignora o se declara como `false`).

removeEventListener(evento, listener**)**—Este método elimina el listener de un elemento.

Los nombres de los eventos que requieren estos métodos no son los mismos que los nombres de los atributos que hemos utilizado hasta el momento. En realidad, los nombres de los atributos se han definido agregando el prefijo on al nombre real del evento. Por ejemplo, el atributo onclick representa el evento click. De la misma manera, tenemos el evento load (onload), el evento mouseover (onmouseover) y así sucesivamente. Cuando usamos el método addEventListener() para hacer que un elemento responda a un evento, tenemos que especificar el nombre real del evento entre comillas, como en el siguiente ejemplo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function agregarevento() {
      var elemento = document.querySelector("section > button");
      elemento.addEventListener("click", mostrarmensaje);
    }
    function mostrarmensaje() {
      alert("Presionó el botón");
    }
    window.addEventListener("load", agregarevento);
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <button type="button">Mostrar</button>
  </section>
</body>
</html>
```

Listado 4-157: *Respondiendo a eventos con el método* `addEventListener()`

El código del Listado 4-157 es el mismo que el del ejemplo anterior, pero ahora utilizamos el método addEventListener() para agregar listeners al objeto Window y el elemento <button>.

5.3. Objetos Event

Cada función que responde a un evento recibe un objeto que contiene información acerca del evento. Aunque algunos eventos tienen sus propios objetos, existe un objeto llamado Event que es común a cada evento. Las siguientes son algunas de sus propiedades y métodos.

target—Esta propiedad devuelve una referencia al objeto que ha recibido el evento (generalmente es un objeto **`Element`**).

type—Esta propiedad devuelve una cadena de caracteres con el nombre del evento.

preventDefault()—Este método cancela el evento para prevenir que el sistema realice tareas por defecto (ver Capítulo 17, Listado 17-3).

stopPropagation()—Este método detiene la propagación del evento a otros elementos, de modo que solo el primer elemento que recibe el evento puede procesarlo (normalmente se aplica a elementos que se superponen y pueden responder al mismo evento).

El objeto Event se envía a la función como un argumento y, por lo tanto, tenemos que declarar un parámetro que recibirá este valor. El nombre del parámetro es irrelevante, pero se define normalmente como e o evento. En el siguiente ejemplo, usamos el objeto Event para identificar el elemento en el que el usuario ha hecho clic.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function agregareventos() {
      var lista = document.querySelectorAll("section > p");
      for (var f = 0; f < lista.length; f++) {
        var elemento = lista[f];
        elemento.addEventListener("click", cambiarcolor);
      }
    }
    function cambiarcolor(evento) {
      var elemento = evento.target;
      elemento.style.backgroundColor = "#999999";
    }
    window.addEventListener("load", agregareventos);
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <p>Mensaje número 1</p>
    <p>Mensaje número 2</p>
    <p>Mensaje número 3</p>
  </section>
</body>
</html>
```

Listado 4-158: *Usando objetos* `Event`

El código del Listado 4-158 agrega un listener para el evento click a cada elemento <p> dentro del elemento <section> de nuestro documento, pero todos se procesan con la misma función. Para identificar en qué elemento ha hecho clic el usuario desde la función, leemos la propiedad target del objeto Event. Esta propiedad devuelve una referencia al objeto Element que representa el elemento que ha recibido el clic. Usando esta referencia, modificamos el fondo del elemento. En consecuencia, cada vez que el usuario hace clic en el área ocupada por un elemento <p>, el fondo de ese elemento se vuelve gris.

Figura 4-8: *Solo afecta al elemento que recibe el evento*

El objeto Event se pasa de forma automática cuando se llama a la función. Si queremos enviar nuestros propios valores junto con este objeto, podemos procesar el evento con una función anónima. La función anónima solo recibe el objeto Event, pero desde el interior de esta función podemos llamar a la función que se encarga de responder al evento con todos los atributos que necesitemos.

```
<script>
  function agregareventos() {
    var lista = document.querySelectorAll("section > p");
    for (var f = 0; f < lista.length; f++) {
      var elemento = lista[f];
      elemento.addEventListener("click", function(evento) {
          var mivalor = 125;
          cambiarcolor(evento, mivalor);
      });
    }
  }
  function cambiarcolor(evento, mivalor) {
    var elemento = evento.target;
    elemento.innerHTML = "Valor " + mivalor;
  }
  window.addEventListener("load", agregareventos);
</script>
```

Listado 4-159: *Respondiendo a un evento con una función anónima*

El código del Listado 4-159 reemplaza al código del ejemplo anterior. Esta vez, en lugar de llamar a la función cambiarcolor() directamente, primero ejecutamos una función anónima. Esta función recibe el objeto Event, declara una nueva variable llamada mivalor con el valor 125, y luego llama a la función cambiarcolor() con ambos valores. Usando estos valores, la función cambiarcolor() modifica el contenido del elemento.

Figura 4-9: *El elemento se modifica con los valores recibidos por la función*

Ejercicio 12:cree un nuevo archivo HTML con el documento del Listado 4-158. Abra el documento en su navegador y haga clic en el área que ocupa el elemento **<p>**. El color de fondo del elemento en el que ha hecho clic debería cambiar a gris. Actualice el código JavaScript con el código del Listado 4-159 y abra el documento nuevamente o actualice la página. Haga clic en un elemento. El contenido de ese elemento se debería reemplazar con el texto "Valor 125", tal como ilustra la Figura 4-9.

En el ejemplo del Listado 4-159, el valor que pasa a la función cambiarcolor() junto con el objeto Event ha sido un valor absoluto (125), pero nos encontraremos con un problema si intentamos pasar el valor de una variable. En este caso, como las instrucciones dentro de la función anónima no se procesan hasta que ocurre el evento, la función contiene una referencia a la variable en lugar de su valor actual. El problema se vuelve evidente cuando trabajamos con valores generados por un bucle.

```
<script>
  function agregareventos() {
    var lista = document.querySelectorAll("section > p");
    for (var f = 0; f < lista.length; f++) {
      var elemento = lista[f];
      elemento.addEventListener("click", function(evento) {
          var mivalor = f;
          cambiarcolor(evento, mivalor);
      });
    }
  }
  function cambiarcolor(evento, mivalor) {
    var elemento = evento.target;
    elemento.innerHTML = "Valor " + mivalor;
  }
  window.addEventListener("load", agregareventos);
</script>
```

Listado 4-160: *Pasando valores a la función que responde al evento*

En este ejemplo, en lugar del valor 125, asignamos la variable f a la variable mivalor, pero debido a que la instrucción no se procesa hasta que el usuario hace clic en el elemento, el sistema asigna una referencia de la variable f a mivalor, no su valor actual. Esto significa que el sistema va a leer la variable f y asignar su valor a la variable mivalor solo cuando el evento click se desencadena, y para entonces el bucle for ya habrá finalizado y el valor actual de f será 3 (el valor final de f cuando el bucle finaliza es 3 porque hay tres elementos <p> dentro del elemento **<section>**). Esto significa que el valor que este código pasa a la función **cambiarcolor()** es siempre 3, sin importar en qué elemento hacemos clic.

Figura 4-10: *El valor pasado a la función es siempre 3*

Este problema se puede resolver combinando dos funciones anónimas. Una función se ejecuta de inmediato, y la otra es la que devuelve la primera. La función principal debe recibir el valor actual de f, almacenarlo en otra variable y luego devolver una segunda función anónima con estos valores. La función anónima devuelta es la que se ejecutará cuando ocurra el evento.

```
<script>
  function agregareventos() {
    var lista = document.querySelectorAll("section > p");
    for (var f = 0; f < lista.length; f++) {
      var elemento = lista[f];
      elemento.addEventListener("click", function(x) {
          return function(evento) {
            var mivalor = x;
            cambiarcolor(evento, mivalor);
          };
      }(f));
    }
  }
  function cambiarcolor(evento, mivalor) {
    var elemento = evento.target;
    elemento.innerHTML = "Valor " + mivalor;
  }
  window.addEventListener("load", agregareventos);
</script>
```

Listado 4-161: *Pasando valores con funciones anónimas*

La función anónima principal se ejecuta cuando se procesa el método addEventListener(). La función recibe el valor actual de la variable f (el valor se pasa a la función por medio de los paréntesis al final de la declaración) y lo asigna a la variable x. Luego la función devuelve una segunda función anónima que asigna el valor de la variable x a mivalor y llama a la función cambiarcolor() para responder al evento. Debido a que el intérprete lee el valor de f en cada ciclo del bucle para poder enviarlo a la función anónima principal, la función anónima que devuelve siempre trabaja con un valor diferente. Para el primer elemento <p>, el valor será 0 (es el primer elemento de la lista y f empieza a contar desde 0), el segundo elemento obtiene un 1 y el tercer elemento un 2. Ahora, el código produce un contenido diferente para cada elemento.

Figura 4-11: *El valor es diferente para cada elemento*

Algunos eventos generan valores únicos que se pasan a la función para ser procesados. Estos eventos trabajan con sus propios tipos de objetos que heredan del objeto Event. Por ejemplo, los eventos del ratón envían un objeto MouseEvent a la función. Estas son algunas de sus propiedades.

button—Esta propiedad devuelve un entero que representa el botón que se ha pulsado (0 = botón izquierdo).

ctrlKey—Esta propiedad devuelve un valor booleano que determina si la tecla Control se ha pulsado cuando ha ocurrido el evento.

altKey—Esta propiedad devuelve un valor booleano que determina si la tecla Alt (Option) se ha pulsado cuando ha ocurrido el evento.

shiftKey—Esta propiedad devuelve un valor booleano que determina si la tecla Shift se ha pulsado cuando ha ocurrido el evento.

metaKey—Esta propiedad devuelve un valor booleano que determina si la tecla Meta se ha pulsado cuando ha ocurrido el evento (la tecla Meta es la tecla Windows en teclados Windows o la tecla Command en teclados Macintosh).

clientX—Esta propiedad devuelve la coordenada horizontal donde estaba ubicado el ratón cuando ha ocurrido el evento. La coordenada se devuelve en píxeles y hace referencia al área que ocupa la ventana.

clientY—Esta propiedad devuelve la coordenada vertical donde estaba ubicado el ratón cuando ha ocurrido el evento. La coordenada se devuelve en píxeles y hace referencia al área que ocupa la ventana.

offsetX—Esta propiedad devuelve la coordenada horizontal donde estaba ubicado el ratón cuando ha ocurrido el evento. La coordenada se devuelve en píxeles y hace referencia al área que ocupa el elemento que ha recibido el evento.

offsetY—Esta propiedad devuelve la coordenada vertical donde estaba ubicado el ratón cuando ha ocurrido el evento. La coordenada se devuelve en píxeles y hace referencia al área que ocupa el elemento que ha recibido el evento.

pageX—Esta propiedad devuelve la coordenada horizontal donde el ratón estaba ubicado cuando ha ocurrido el evento. La coordenada se devuelve en píxeles y hace relación al documento. El valor incluye el desplazamiento del documento.

pageY—Esta propiedad devuelve la coordenada vertical donde el ratón estaba ubicado cuando ha ocurrido el evento. La coordenada se devuelve en píxeles y hace relación al documento. El valor incluye el desplazamiento del documento.

screenX—Esta propiedad devuelve la coordenada horizontal donde el ratón estaba ubicado cuando ha ocurrido el evento. La coordenada se devuelve en píxeles y hace relación a la pantalla.

screenY—Esta propiedad devuelve la coordenada vertical donde el ratón estaba ubicado cuando ha ocurrido el evento. La coordenada se devuelve en píxeles y hace relación a la pantalla.

movementX—Esta propiedad devuelve la diferencia entre la posición actual y la anterior del ratón en el eje horizontal. El valor se devuelve en píxeles y hace relación a la pantalla.

movementY—Esta propiedad devuelve la diferencia entre la posición actual y la anterior del ratón en el eje vertical. El valor se devuelve en píxeles y hace relación a la pantalla.

En el Capítulo 3 hemos explicado que la pantalla está dividida en filas y columnas de píxeles, y los ordenadores usan un sistema de coordenadas para identificar la posición de cada píxel (ver Figura 3-50). Lo que no hemos mencionado es que este mismo sistema de coordenadas se aplica a cada área, incluida la pantalla, la ventana del navegador, y los elementos HTML, por lo que cada uno de ellos tiene su propio origen (sus coordenadas siempre comienzan en el punto 0, 0). El objeto MouseEvent nos da las coordenadas del ratón cuando ha ocurrido el evento, pero debido a que cada área tiene su propio sistema de coordenadas, se obtienen diferentes valores. Por ejemplo, las propiedades clientX y clientY contienen las coordenadas del ratón en el sistema de coordenadas de la ventana, pero las propiedades offsetX y offsetY informan de la posición en el sistema de coordenadas del elemento que recibe el evento. El siguiente ejemplo detecta un clic y muestra la posición del ratón dentro de la ventana mediante las propiedades clientX y clientY.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function mostrarposicion(evento) {
      alert("Posicion: " + evento.clientX + " / " + evento.clientY);
    }
    window.addEventListener("click", mostrarposicion);
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <p>Este es mi sitio web</p>
  </section>
</body>
</html>
```

Listado 4-162: *Información la posición del ratón*

El código del Listado 4-162 responde al evento click en el objeto Window, por lo que un clic en cualquier parte de la ventana ejecutará la función mostrarposicion() y la posición del ratón se mostrará en la pantalla. Esta función lee las propiedades clientX y clientY para obtener la posición del ratón relativa a la pantalla. Si queremos obtener la posición relativa a un elemento, tenemos que responder al evento desde el elemento y leer las propiedades **`offsetX`** y **`offsetY`**. El siguiente ejemplo usa estas propiedades para crear una barra de progreso cuyo tamaño está determinado por la posición actual del ratón cuando está sobre el elemento.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <style>
    #contenedor {
      width: 500px;
      height: 40px;
      padding: 10px;
      border: 1px solid #999999;
    }

    #barraprogreso {
      width: 0px;
      height: 40px;
      background-color: #000099;
    }
  </style>
  <script>
    function iniciar() {
      var elemento = document.getElementById("contenedor");
      elemento.addEventListener("mousemove", moverbarra);
    }
    function moverbarra(evento) {
      var anchobarra = evento.offsetX - 10;
      if (anchobarra < 0) {
        anchobarra = 0;
      } else if (anchobarra > 500) {
        anchobarra = 500;
      }
      var elemento = document.getElementById("barraprogreso");
      elemento.style.width = anchobarra + "px";
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <h1>Nivel</h1>
    <div id="contenedor">
      <div id="barraprogreso"></div>
    </div>
  </section>
</body>
</html>
```

Listado 4-163: *Calculando la posición del ratón en un elemento*

El documento del Listado 4-163 incluye dos elementos , uno dentro del otro, para recrear una barra de progreso. El elemento identificado con el nombre contenedor trabaja como un contenedor para establecer los límites de la barra, y el identificado con el nombre barraprogreso representa la barra misma. El propósito de esta aplicación es permitir al usuario determinar el tamaño de la barra con el ratón, por lo que tenemos que responder al evento mousemove para seguir cada movimiento del ratón y leer la propiedad offsetX para calcular el tamaño de la barra basado en la posición actual.

Debido a que el área ocupada por el elemento barraprogreso siempre será diferente (se define con un ancho de 0 píxeles por defecto), tenemos que responder al evento mousemove desde el elemento contenedor. Esto requiere que el código ajuste los valores que devuelve la propiedad offsetX a la posición del elemento barraprogreso. El elemento contenedor incluye un relleno de 10 píxeles, por lo que la barra estará desplazada 10 píxeles desde el lado izquierdo de su contenedor, y ese es el número que debemos restar al valor de offsetX para determinar el ancho de la barra (event.offsetX - 10). Por ejemplo, si el ratón está ubicado a 20 píxeles del lado izquierdo del contenedor, significa que se encuentra solo a 10 píxeles del lado izquierdo de la barra, por lo que la barra debería tener un ancho de 10 píxeles. Esto funciona hasta que el ratón se ubica sobre el relleno del elemento contenedor. Cuando el ratón se localiza sobre el relleno izquierdo, digamos en la posición 5, la operación devuelve el valor -5, pero no podemos declarar un tamaño negativo para la barra. Algo similar pasa cuando el ratón se sitúa sobre el relleno derecho. En este caso, la barra intentará sobrepasar el tamaño máximo del contenedor. Estas situaciones se resuelven con las instrucciones if. Si el nuevo ancho es menor a 0, lo declaramos como 0, y si es mayor de 500, lo declaramos como 500. Con estos límites establecidos, obtenemos una referencia al elemento barraprogreso y modificamos su propiedad width para declarar el nuevo ancho.

Figura 4-12: *Barra de progreso*

Ejercicio 13:cree un nuevo archivo HTML con el documento del Listado 4-163 y abra el documento en su navegador. Mueva el ratón sobre el elemento **`contenedor`**. Debería ver el elemento **`barraprogreso`** expandirse o encogerse siguiendo el ratón, tal como muestra la Figura 4-12.

Otros eventos que producen sus propios objetos Event son los que están relacionados con el teclado (keypress, keydown, y keyup). El objeto es de tipo KeyboardEvent e incluye las siguientes propiedades.

key—Esta propiedad devuelve una cadena de caracteres que identifica la tecla o las teclas que han desencadenado el evento.

ctrlKey—Esta propiedad devuelve un valor booleano que determina si se ha pulsado la tecla Control cuando ha ocurrido el evento.

altKey—Esta propiedad devuelve un valor booleano que determina si se ha pulsado la tecla Alt (Option) cuando ha ocurrido el evento.

shiftKey—Esta propiedad devuelve un valor booleano que determina si se ha pulsado la tecla Shift cuando ha ocurrido el evento.

metaKey—Esta propiedad devuelve un valor booleano que determina si se ha pulsado la tecla Meta cuando ha ocurrido el evento (la tecla Meta es la tecla Windows en los teclados Windows o la tecla Command en los teclados Macintosh).

repeat—Esta propiedad devuelve un valor booleano que determina si el usuario pulsa la tecla continuamente.

La propiedad más importante del objeto KeyboardEvent es key. Esta propiedad devuelve una cadena de caracteres que representa la tecla que ha desencadenado el evento. Las teclas comunes como los números y letras producen una cadena de caracteres con los mismos caracteres en minúsculas. Por ejemplo, si queremos comprobar si la tecla pulsada ha sido la letra A, tenemos que comparar el valor con el texto "a". El siguiente ejemplo compara el valor que devuelve la propiedad key con una serie de números para comprobar si la tecla pulsada ha sido 0, 1, 2, 3, 4, o 5.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <style>
    section {
      text-align: center;
    }
    #bloque {
      display: inline-block;
      width: 150px;
      height: 150px;
      margin-top: 100px;
      background-color: #990000;
    }
  </style>
  <script>
    function detectartecla(evento) {
      var elemento = document.getElementById("bloque");
      var 333ódigo = evento.key;
      switch (333ódigo) {
        case "0":
          elemento.style.backgroundColor = "#990000";
          break;
        case "1":
          elemento.style.backgroundColor = "#009900";
          break;
        case "2":
          elemento.style.backgroundColor = "#000099";
          break;
        case "3":
          elemento.style.backgroundColor = "#999900";
          break;
        case "4":
          elemento.style.backgroundColor = "#009999";
          break;
        case "5":
          elemento.style.backgroundColor = "#990099";
          break;
      }
    }
    window.addEventListener("keydown", detectartecla);
  </script>
</head>
<body>
  <section>
    <div id="bloque"></div>
  </section>
</body>
</html>
```

Listado 4-164: *Detectando la tecla pulsada*

El documento del Listado 4-164 dibuja un bloque rojo en el centro de la ventana. Para responder al teclado, agregamos un listener para el evento keydown a la ventana (el evento keydown se desencadena con todas las teclas, mientras que el evento keypress solo se desencadena con teclas comunes, como letras y números). Cada vez que se pulsa una tecla, leemos el valor de la propiedad key y lo comparamos con una serie de números. Si una encuentra una coincidencia, asignamos un color diferente al fondo del elemento. En caso contrario, el código no hace nada.

Además de las teclas comunes, la propiedad key también informa de teclas especiales como Alt o Control. Las cadenas de caracteres generadas por las teclas más comunes son “Alt”, “Control”, “Shift”, “Meta”, “Enter”, “Tab”, “Backspace”, “Delete”, “Escape”, “ “ (barra espaciadora), “ArrowUp”, “ArrowDown”, “ArrowLeft”, “ArrowRight”, “Home”, “End”, “PageUp”, y “PageDown”. El siguiente código detecta si las flechas se pulsan para cambiar el tamaño del bloque creado en el ejemplo anterior.

```
<!DOCTYPE html>
<html lang=”es”>
<head>
  <meta charset=”utf-8”>
  <title>JavaScript</title>
  <style>
    section {
      text-align: center;
    }
    #bloque {
      display: inline-block;
      width: 150px;
      height: 150px;
      margin-top: 100px;
      background-color: #990000;
    }
  </style>
  <script>
    function detectartecla(evento) {
      var elemento = document.getElementById(“bloque”);
      var ancho = elemento.clientWidth;
      var 334ódigo = evento.key;
      switch (335ódigo) {
        case “ArrowUp”:
          ancho += 10;
          break;
        case “ArrowDown”:
          ancho -= 10;
          break;
      }
      if (ancho < 50) {
        ancho = 50;
      }
      elemento.style.width = ancho + “px”;
      elemento.style.height = ancho + “px”;
    }
    window.addEventListener(“keydown”, detectartecla);
  </script>
</head>
<body>
  <section>
    <div id=”bloque”></div>
  </section>
</body>
</html>
```

Listado 4-165: *Detectando teclas especiales*

Como hemos hecho en el ejemplo del Listado 4-146, obtenemos el ancho actual del elemento desde la propiedad clientWidth y luego asignamos el nuevo valor a la propiedad width del objeto Styles. El nuevo valor depende de la tecla que se ha pulsado. Si la techa es la flecha hacia arriba, incrementamos el tamaño en 10 píxeles, pero si la tecla es la flecha hacia abajo, el tamaño se reduce en 10 píxeles. Al final, controlamos este valor para asegurarnos de que el bloque no se reduce a menos de 50 píxeles.

Ejercicio 14:cree un nuevo archivo HTML con el documento del Listado 4-165. Abra el documento en su navegador y pulse las flechas hacia arriba y hacia abajo. El bloque debería expandirse o encogerse según la tecla pulsada.

Capítulo 6
Depuración

6.1. Depuración

La depuración (o debugging en inglés) es el proceso de encontrar y corregir los errores en nuestro código. Existen varios tipos de errores, desde errores de programación hasta errores lógicos, e incluso errores personalizados generados para indicar un problema detectado por el mismo código). Algunos errores requieren del uso de herramientas para encontrar una solución y otros solo exigen un poco de paciencia y perseverancia. La mayoría de las veces, para determinar qué es lo que no funciona en nuestro código es necesario leer las instrucciones una por una y seguir la lógica hasta detectar el error. Afortunadamente, los navegadores ofrecen herramientas para ayudarnos a resolver estos problemas, y JavaScript incluye algunas técnicas que podemos implementar para facilitar este trabajo.

6.2. Consola

La herramienta más útil para controlar errores y corregir nuestro código es la consola. Las consolas están disponibles en casi todos los navegadores, pero de diferente formas. Generalmente, se abren en la parte inferior de la ventana del navegador y están formadas por varios paneles que detallan información de cada aspecto del documento, incluidos el código HTML, los estilos CSS y, por supuesto, JavaScript. El panel Console es el que muestra los errores y mensajes personalizados.

Figura 4-13: *Consola de Google Chrome*

Lo básico: el acceso a esta consola varía de un navegador a otro, e incluso entre diferentes versiones de un mismo navegador, pero las opciones se encuentran normalmente en el menú principal con el nombre de Herramientas de desarrollo u otras herramientas.

Los tipos de errores que vemos a menudo impresos en la consola son errores de programación. Por ejemplo, si llamamos a una función inexistente o tratamos de leer una propiedad que no es parte del objeto, se considera un error de programación y se informa de él en la consola.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    funcionfalsa();
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
  </section>
</body>
</html>
```

Listado 4-166: *Generando un error*

En el Listado 4-166 hemos intentado ejecutar una función llamada funcionfalsa() que no se había definido previamente. El navegador encuentra el error y muestra el mensaje "funcionfalsa is not defined" ("funcionfalsa no está definida") en la consola para reportarlo.

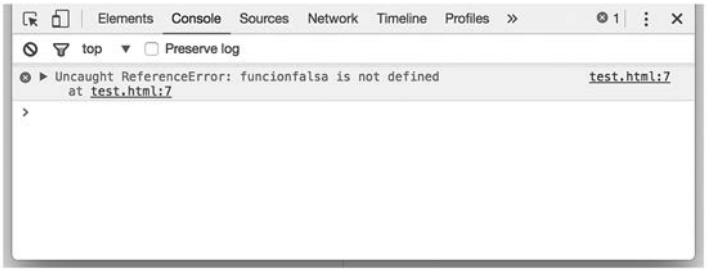


Figura 4-14: *Error reportado en la consola*

Ejercicio 15:cree un nuevo archivo HTML con el documento del Listado 4-166 y abra el documento en su navegador. Acceda al menú principaldel navegador y busque la opción para abrir la consola. En Google Chrome, el menú se encuentra en la esquina superior derecha y la opción se denomina Más herramientas/Herramientas de desarrollo. Debería ver el error que genera la función impreso en la consola, tal como ilustra la Figura 4-14.

6.3. Objeto Console

Como ya hemos mencionado, a veces los errores no son errores de programación, sino errores lógicos. El intérprete JavaScript no puede encontrar ningún error en el código, pero la aplicación no hace lo que esperamos. Esto se puede deber a varios motivos, desde una operación que olvidamos realizar, hasta una variable iniciada con el valor incorrecto. Estos son los errores más difíciles de identificar, pero existe una técnica de programación tradicional llamada breakpoints (puntos de interrupción) que puede ayudarnos a encontrar una solución. Los breakpoints son puntos de interrupción en nuestro código que establecemos para controlar el estado actual de la aplicación. En un breakpoint, mostramos los valores actuales de las variables o un mensaje que nos informa de que el intérprete ha llegado a esa parte del código.

Tradicionalmente, los programadores de JavaScript insertaban un método alert() en partes del código para exponer valores que los ayudaran a encontrar el error, pero este método no es apropiado en la mayoría de las situaciones porque detiene la ejecución del código hasta que se cierra la ventana emergente. Los navegadores simplifican este proceso creando un objeto de tipo Console. Este objeto se asigna a la propiedad console del objeto Window y se transforma en la conexión entre nuestro código y la consola del navegador. Los siguientes son algunos de los métodos que se incluyen en el objeto Console para manipular la consola.

Log(valor**)**—Este método muestra el valor entre paréntesis en la consola.

Assert(condición, valores**)**—Este método muestra en la consola los valores que especifican los atributos si la condición especificada por el primer atributo es falsa.

Clear()—Este método limpia la consola. Los navegadores también ofrecen un botón en la parte superior de la consola con la misma funcionalidad.

El método más importante en el objeto Console es log(). Con este método podemos imprimir un mensaje en la consola en cualquier momento, sin interrumpir la ejecución del código, lo cual significa que podemos controlar los valores de las variables y propiedades cada vez que lo necesitemos y ver si cumplen con nuestras expectativas. Por ejemplo, podemos imprimir en la consola los valores generados por un bucle en cada ciclo para asegurarnos de que estamos creando la secuencia correcta de números.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    var lista = [0, 5, 103, 24, 81];
    for(var f = 0; f < lista.length; f++) {
      console.log("El valor actual es " + f);
    }
    console.log("El valor final de f es: " + f);
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
  </section>
</body>
</html>
```

Listado 4-167: *Mostrando mensajes en la consola con el método* `log()`

El código del Listado 4-167 llama al método log() para mostrar un mensaje en cada ciclo del bucle y también al final para mostrar el último valor de la variable f, por lo que se imprimen un total de cinco mensajes en la consola.

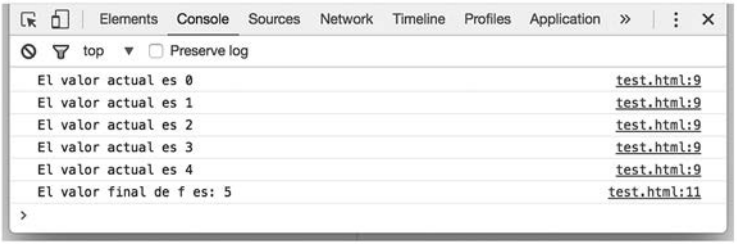


Figura 4-15: *Mensajes en la consola*

Este breve ejemplo ilustra el poder del método log() y cómo nos puede ayudar a entender la forma en la que trabaja nuestro código. En este caso, muestra el mecanismo de un bucle for. El valor de la variable f en el bucle oscila entre 0 y un número menos que la cantidad de valores en el array (5), por lo que las instrucciones dentro del bucle imprimen un total de cinco mensajes con los valores 0, 1, 2, 3, y 4. Esto es lo que se espera, pero el método log() al final del código imprime el valor final de f, que no es 4 sino 5. En el primer ciclo del bucle, el intérprete comprueba la condición con el valor inicial de f. Si la condición es verdadera, ejecuta el código. Pero en el siguiente ciclo, el intérprete ejecuta la operación asignada al bucle (f++) antes de comprobar la condición. Si la condición es falsa, el bucle se interrumpe. Esta es la razón por la que el valor final de f es 5. Al final del bucle, el valor de f se ha incrementado una vez más antes de que la condición se haya comprobado.

6.4. Evento error

En ciertos momentos, nos encontraremos con errores que no hemos generado nosotros. A medida que nuestra aplicación crece e incorpora librerías sofisticadas y API, los errores comienzan a depender de factores externos, como recursos que se vuelven inaccesibles o cambios inesperados en el dispositivo que está ejecutando nuestra aplicación. Con el propósito de ayudar al código a detectar estos errores y corregirse a sí mismo, JavaScript ofrece el evento error. Este evento está disponible en varias API, como veremos más adelante, pero también como un evento global al que podemos responder desde el objeto Window.

Al igual que otros eventos, el evento error crea su propio objeto Event llamado ErrorEvent, para transmitir información a la función. Este objeto incluye las siguientes propiedades.

Error—Esta propiedad devuelve un objeto con información sobre el error.

Message—Esta propiedad devuelve una cadena de caracteres que describe el error.

Lineno—Esta propiedad devuelve la línea en el documento donde ha ocurrido el error.

Colno—Esta propiedad devuelve la columna donde comienza la instrucción que ha producido el error.

Filename—Esta propiedad devuelve la URL del archivo donde ha ocurrido el error.

Con este evento, podemos programar nuestro código para que responda a errores inesperados.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function mostrarerror(evento){
      console.log('Error: ' + evento.error);
      console.log('Mensaje: ' + evento.message);
      console.log('Línea: ' + evento.lineno);
      console.log('Columna: ' + evento.colno);
      console.log('URL: ' + evento.filename);
    }
    window.addEventListener('error', mostrarerror);
    funcionfalsa();
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
  </section>
</body>
</html>
```

Listado 4-168: Respondiendo a errores

En el código del Listado 4-168, el error se produce por la ejecución de una función inexistente llamada funcionfalsa(). Cuando el navegador intenta ejecutar esta función, encuentra el error y dispara el evento error para informar de él. Para identificar el error, imprimimos mensajes en la consola con los valores de las propiedades del objeto ErrorEvent.

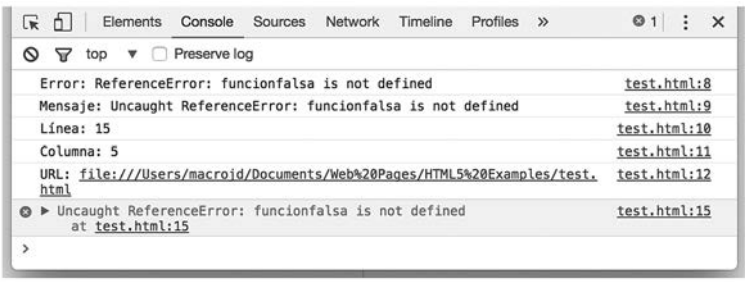


Figura 4-16: *Información acerca del error*

6.5. Excepciones

A veces sabemos de antemano que nuestro código puede producir un error. Por ejemplo, podemos tener una función que calcula un número a partir de un valor insertado por el usuario. Si el valor recibido se encuentra fuera de cierto rango, la operación no será válida. En programación, los errores que se pueden gestionar por el código se llaman excepciones, y el proceso de generar una excepción se llama arrojar (throw). En estos términos, cuando informamos de nuestros propios errores decimos que arrojamos una excepción. JavaScript incluye las siguientes instrucciones para arrojar excepciones y capturar errores.

Throw—Esta instrucción genera una excepción.

Try—Esta instrucción indica el grupo de instrucciones que pueden generar errores.

Catch—Esta instrucción indica el grupo de instrucciones que deberían ejecutarse si ocurre una excepción.

Si sabemos que una función puede producir un error, podemos detectarlo, arrojar una excepción con la instrucción throw, y luego responder a la excepción con la combinación de las instrucciones try y catch. El siguiente ejemplo ilustra cómo funciona este proceso.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    var existencia = 5;
    function vendido(cantidad) {
      if (cantidad > existencia) {
        var error = {
          name: "ErrorExistencia",
          message: "Sin Existencia"
        };
        throw error;
      } else {
        existencia = existencia - cantidad;
      }
    }
    try {
      vendido(8);
    } catch(error) {
      console.log(error.message);
    }
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
  </section>
</body>
</html>
```

Listado 4-169: Arrojando excepciones

La instrucción throw trabaja de modo similar a la instrucción return; detiene la ejecución de la función y devuelve un valor que se captura mediante la instrucción catch. El valor se debe especificar como un objeto con las propiedades name y message. La propiedad name debería tener un valor que identifique la excepción y la propiedad message debería contener un mensaje que describa el error. Una vez que tenemos una función que arroja una excepción, tenemos que llamarla desde las instrucciones try catch. La sintaxis de estas instrucciones es similar a la de las instrucciones if else. Estas instrucciones definen dos bloques de código. Si las instrucciones dentro del bloque try arrojan una excepción, se ejecutan las instrucciones dentro del bloque catch.

En nuestro ejemplo, hemos creado una función llamada vendido() que lleva la cuenta de los ítems vendidos en una tienda. Cuando un cliente realiza una compra, llamamos a esta función con el número de ítems vendidos. La función recibe este valor y lo resta de la variable existencia. En este punto es donde controlamos si la transacción es válida. Si no hay existencias suficiente para satisfacer el pedido, arrojamos una excepción. En este caso, la variable existencia se inicializa con el valor 5 y la función vendido() se llama con el valor 8, por lo tanto la función arroja una excepción. Debido a que la llamada se realiza dentro de un bloque try, la excepción se captura, las instrucciones dentro del bloque catch se ejecutan y en la consola se muestra el mensaje “Sin Existencia”.

Capítulo 7
API

7.1. API

Por más experiencia o conocimiento que tengamos sobre programación de ordenadores y el lenguaje de programación que usamos para crear nuestras aplicaciones, nunca podremos programar la aplicación completa por nuestra cuenta. Crear un sistema de base de datos o generar gráficos complejos en la pantalla nos llevaría una vida entera si no contáramos con la ayuda de otros programadores y desarrolladores. En programación, esa ayuda se facilita en forma de librerías y API. Una librería es una colección de variables, funciones y objetos que realizan tareas en común, como calcular los píxeles que el sistema tiene que activar en la pantalla para mostrar un objeto tridimensional o filtrar los valores que devuelve una base de datos. Las librerías reducen la cantidad de código que un desarrollador tiene que escribir y ofrecen soluciones estándar que funcionan en todos los navegadores. Debido a su complejidad, las librerías siempre incluyen una interfaz, un grupo de variables, funciones y objetos que podemos usar para comunicarnos con el código y describir lo que queremos que la librería haga por nosotros. Esta parte visible de la librería se denomina API (del inglés, Application Programming Interface) y es lo que en realidad tenemos que aprender para poder incluir la librería en nuestros proyectos.

7.2. Librerías nativas

Lo que convirtió a HTML5 en la plataforma de desarrollo líder que es actualmente no fueron las mejoras introducidas en el lenguaje HTML, o la integración entre este lenguaje con CSS y JavaScript, sino la definición de un camino a seguir para la estandarización de las herramientas que las empresas facilitan por defecto en sus navegadores. Esto incluye un grupo de librerías que se encargan de tareas comunes como la generación de gráficos 2D y 3D, almacenamiento de datos, comunicaciones y mucho más. Gracias a HTML5, ahora los navegadores incluyen librerías eficaces con API integradas en objetos JavaScript y, por lo tanto, disponibles para nuestros documentos. Implementando estas API en nuestro código, podemos ejecutar tareas complejas con solo llamar un método o declarar el valor de una propiedad.

IMPORTANTE: las API nativas se han convertido en una parte esencial del desarrollo de aplicaciones profesionales y videojuegos y, por lo tanto, se transformarán en nuestro objeto de estudio de aquí en adelante.

7.3. Librerías externas

Antes de la aparición de HTML5, se desarrollaron varias librerías programadas en JavaScript para superar las limitaciones de las tecnologías disponibles en el momento. Algunas se crearon con propósitos específicos, desde procesar y validar formularios hasta la generación y manipulación de gráficos. A través de los años, algunas de estas librerías se han vuelto extremadamente populares, y algunas de ellas, como Google Maps, son imposibles de imitar por desarrolladores independientes.

Estas librerías no son parte de HTML5, pero constituyen un aspecto importante del desarrollo web, y algunas de ellas se han implementado en los sitios web y aplicaciones más destacados de la actualidad. Aprovechan todo el potencial de JavaScript y contribuyen al desarrollo de nuevas tecnologías para la Web. La siguiente es una lista de las más populares.

- **jQuery** (www.jquery.com) es una librería multipropósito que simplifica el código JavaScript y la interacción con el documento. También facilita la selección de elementos HTML, la generación de animaciones, el control de eventos y la implementación de Ajax en nuestras aplicaciones.
- **React** (facebook.github.io/react) es una librería gráfica que nos ayuda a crear interfaces de usuario interactivas.
- **AngularJS** (www.angularjs.org) es una librería que expande los elementos HTML para volverlos más dinámicos e interactivos.
- **Node.js** (www.nodejs.org) es una librería que funciona en el servidor y tiene el propósito de construir aplicaciones de red.
- **Modernizr** (www.modernizr.com) es una librería que puede detectar características disponibles en el navegador, incluidas propiedades CSS, elementos HTML y las API de JavaScript.
- **Moment.js** (www.momentjs.com) es una librería cuyo único propósito es procesar fechas.
- **Three.js** (www.threejs.org) es una librería de gráficos 3D basada en una API incluida en los navegadores llamada WebGL (Web Graphics Library). Estudiaremos esta librería y WebGL en el Capítulo 12.
- **Google Maps** (developers.google.com/maps/) es un grupo de librerías diseñadas para incluir mapas en nuestros sitios web y aplicaciones.

Estas librerías suelen ser pequeños archivos que podemos descargar de un sitio web e incluir en nuestro documentos con el elemento <script>, como lo hacemos con nuestros propios archivos JavaScript. Una vez que se incluye la librería en el documento, podemos acceder a su API desde nuestro código. Por ejemplo, el siguiente documento incluye la librería Modernizr para detectar si la propiedad CSS box-shadow está o no disponible en el navegador del usuario.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Modernizr</title>
  <script src="modernizr-custom.js"></script>
  <script>
    function iniciar(){
      var elemento = document.getElementById("subtitulo");
      if (Modernizr.boxshadow) {
        elemento.innerHTML = 'Box Shadow está disponible';
      } else {
        elemento.innerHTML = 'Box Shadow no está disponible';
      }
    }
    window.addEventListener('load', iniciar);
  </script>
</head>
<body>
  <section>
    <h1>Sitio Web</h1>
    <p id="subtitulo"></p>
  </section>
</body>
</html>
```

Listado 4-170: *Detectando funciones con Modernizr*

Modernizr crea un objeto llamado Modernizr que ofrece propiedades por cada característica de HTML5 que queremos detectar. Estas propiedades devuelven un valor booleano que será true o false dependiendo de si la característica está disponible o no. Para incluir esta librería, tenemos que descargar el archivo desde su sitio web (www.modernizr.com) y luego agregarlo a nuestro documento con el elemento <script>, como hemos hecho en el Listado 4-170.

El archivo generado por el sitio web se denomina modernizr-custom.js y contiene un sistema de detección para todas las características que hemos seleccionado. En nuestro ejemplo, seleccionamos la característica Box Shadow porque eso es lo único que queremos verificar. Una vez que se carga la librería, tenemos que leer el valor de la propiedad que representa la característica y responder de acuerdo al resultado. En este caso, insertamos un texto en un elemento <p>. Si la propiedad box-shadow está disponible, el elemento mostrará el mensaje "Box Shadow está disponible "; de lo contrario, el mensaje que muestra el elemento será "Box Shadow no está disponible".

Ejercicio 16:cree un nuevo archivo HTML con el documento del Listado 4-170. Vaya a www.modernizr.com, seleccione la característica Box Shadow (o las que quiera verificar), y haga clic en Build para crear su archivo. Se descargará archivo llamado modernizr-custom.js en su ordenador. Mueva el archivo al directorio de su documento y abra el documento en su navegador. Si su navegador soporta la propiedad CSS **`box-shadow`**, debería ver el mensaje "Box Shadow está disponible" en la pantalla.

IMPORTANTE: existen docenas de librerías externas programadas en JavaScript. Este libro no desarrolla el tema, pero puede visitar nuestro sitio web y seguir los enlaces de este capítulo para obtener más información.