Capítulo 6
JavaScript

6.1 Introducción a JavaScript

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.

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 6-1: *Definiendo JavaScript en línea*

El atributo **onclick** agregado al elemento **<p>** del Listado 6-1 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 6-1: *Ventana emergente generada por la función* `alert()`

Hágalo usted mismo: cree un nuevo archivo HTML con el código del Listado 6-1. 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 6-1.

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 6-1, 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 6-2: *Implementando múltiples atributos de evento*

Hágalo usted mismo: actualice su archivo HTML con el código del Listado 6-2. 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 6-3: *Respondiendo al evento* `load`

El documento del Listado 6-3 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 6-3 en su navegador, verá que la 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 6-4: *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.

Hágalo usted mismo: actualice su archivo HTML con el código del Listado 6-4 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 pulsamos el 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 6-5: Introduciendo código JavaScript desde un archivo externo

El elemento **<script>** del Listado 6-5 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 6-6: Creando un archivo JavaScript (micodigo.js)

Hágalo usted mismo: actualice su archivo HTML con el código del Listado 6-5. Cree un nuevo archivo con el nombre micodigo.js y el código JavaScript del Listado 6-6. 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.

Variables

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.

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 6-7: *Declarando una variable en JavaScript*

La instrucción del Listado 6-7 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 6-8: *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 6-8, 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 6-2: *Ventana emergente mostrando el valor de una variable*

Hágalo usted mismo: actualice su archivo micodigo.js con el código del Listado 6-8. Abra el documento del Listado 6-5 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 6-9: *Asignando un nuevo valor a la variable*

En el Listado 6-9, 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 6-10: *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 6-11: *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 6-12: *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 6-12, 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 6-13: *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 6-14: *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 6-15: *Incrementando el valor de una variable en un valor específico*

El proceso generado por el código del Listado 6-15 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 6-16: *Calculando el resto de una división*

La operación asignada a la variable **minumero** del Listado 6-16 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 6-17: *Determinando la paridad de un número*

El código del Listado 6-17 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.

Hágalo usted mismo: 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.

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 6-18: *Asignando una cadena de caracteres a una variable*

El código del Listado 6-18 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 6-3: *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 6-19: *Concatenando texto*

El código del Listado 6-19 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 6-20: *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 6-21: *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 6-22: *Concatenando números*

Lo básico: el resultado del código del Listado 6-22 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 6-23: *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 6-24: *Generando nuevas líneas de texto*

El código del Listado 6-24 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.

Lo básico: en JavaScript las cadenas de caracteres se declaran como objetos y, por lo tanto, incluyen métodos para realizar operaciones en sus caracteres. Estudiaremos objetos, los objetos **String**, y cómo implementar sus métodos más adelante en este capítulo.

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 6-25: *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 6-26: *Creando arrays*

En el Listado 6-26, 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 6-27: *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 6-27 combinando números y cadenas de caracteres.

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

Listado 6-28: *Almacenando diferentes tipos de valores*

Hágalo usted mismo: reemplace el código en su archivo micodigo.js por el código del Listado 6-28 y abra el documento del Listado 6-5 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 6-29: *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 6-30: *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 6-31: *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 6-32: *Definiendo arrays multidimensionales*

El ejemplo del Listado 6-32 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 6-33: *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 6-34: *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 6-35: *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.

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 6-36: *Comprobando una condición con* `if`

En el código del Listado 6-36, 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 6-37: *Invirtiendo el resultado de la condición*

El código del Listado 6-37 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 6-38: *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 6-39: *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 6-40: *Usando números enteros como condiciones*

El código de la instrucción **if** del Listado 6-40 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 6-41: *Usando cadenas de caracteres como condiciones*

Hágalo usted mismo: reemplace el código en su archivo micodigo.js con el código del Listado 6-41 y abra el documento del Listado 6-5 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 6-42: *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 6-43: *Comprobando un valor con la instrucción* `switch`

En el ejemplo del Listado 6-43, 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 6-44: *Creando un bucle con la instrucción* `for`

En el código del Listado 6-44 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 6-44, 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 6-45: *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 6-45, 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 6-46: *Usando la instrucción* `while`

El ejemplo del Listado 6-46 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 6-47: *Usando la instrucción* `do while`

En el ejemplo del Listado 6-47, 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).

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 6-48: *Saltando hacia el siguiente ciclo del bucle*

La instrucción **if** dentro del bucle **for** del Listado 6-48 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 6-49: *Interrumpiendo el bucle*

Nuevamente, la instrucción **if** del Listado 6-49 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**.

6.2 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.

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 6-50: *Declarando funciones*

Las funciones se deben primero declarar y luego ejecutar. El código del Listado 6-50 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 6-51: *Procesando datos con funciones*