CONTENIDOS WEB

El gran libro de HTML5, CSS3 y JavaScript

J.D Gauchat

3ª Edición

- Modelos de Caja Tradicional y Flexible
- Diseño Web Adaptable
- Vídeo y Audio

El gran libro de HTML5,
CSS3 y JavaScript

Acceda a www.marcombo.info
para descargar gratis
el contenido adicional
complemento imprescindible de este libro

Código: **HTML5**

El gran libro de HTML5, CSS3 y JavaScript

3a edición

J.D Gauchat

Edición original publicada en inglés por Mink Books con el título: *HTML5 for Masterminds*, © J.D Gauchat 2017.

Título de la edición en español:

El gran libro de HTML5, CSS3 y JavaScript

Tercera edición en español, año 2017

© 2017 MARCOMBO, S.A.
Gran Via de les Corts Catalanes, 594
08007 Barcelona
www.marcombo.com

«Cualquier forma de reproducción, distribución, comunicación pública o transformación de esta obra solo puede ser realizada con la autorización de sus titulares, salvo excepción prevista por la ley. Diríjase a CEDRO (Centro Español de Derechos Reprográficos, www.cedro.org) si necesita fotocopiar o escanear algún fragmento de esta obra».

ISBN: 978-84-267-2463-2

D.L.: B-12571-2017

Printed in Spain

Tabla de contenidos

Capítulo 1—Desarrollo web

1.1 Sitios Web **1**
Archivos 1
Dominios y URL 3
Hipervínculos 4
URL absolutas y relativas 5

1.2 Lenguajes **5**
HTML 6
CSS 7
JavaScript 8
Lenguajes de servidor 9

1.3 Herramientas **9**
Editores 10
Registro de dominios 12
Alojamiento web 13
Programas FTP 14
MAMP 16

Capítulo 2—HTML

2.1 Estructura **19**
Tipo de documento 19
Elementos estructurales 20
Atributos globales 32

2.2 Contenido **33**
Texto 34
Enlaces 40
Imágenes 45
Listados 47
Tablas 52
Atributos globales 54

2.3 Formularios **56**
Definición 56
Elementos 57
Enviando el formulario 73
Atributos globales 75

Capítulo 3—CSS

3.1 Estilos **83**
Aplicando estilos 84
Hojas de estilo en cascada 86

3.2 Referencias **87**
Nombres 88
Atributo Id 91
Atributo Class 92

Otros atributos 93
Seudoclases 94
3.3 Propiedades **98**
Texto 98
Colores 103
Tamaño 105
Fondo 110
Bordes 113
Sombras 119
Gradientes 122
Filtros 127
Transformaciones 128
Transiciones 134
Animaciones 136

Capítulo 4—Diseño web
4.1 Cajas **139**
Display 139
4.2 Modelo de caja tradicional **141**
Contenido flotante 141
Cajas flotantes 146
Posicionamiento absoluto 150
Columnas 155
Aplicación de la vida real 158
4.3 Modelo de caja flexible **171**
Contenedor flexible 171
Elementos flexibles 172
Organizando elementos flexibles 179
Aplicación de la vida real 191

Capítulo 5—Diseño web adaptable
5.1 Web móvil **199**
Media Queries 199
Puntos de interrupción 202
Áreas de visualización 204
Flexibilidad 205
Box-sizing 207
Fijo y flexible 208
Texto 214
Imágenes 217
Aplicación de la vida real 224

Capítulo 6—JavaScript
6.1 Introducción a JavaScript **241**
Implementando JavaScript 241
Variables 247
Cadenas de texto 251
Booleanos 253

Arrays 253
Condicionales y bucles 256
Instrucciones de transferencia de control 262
6.2 Funciones **263**
Declarando funciones 263
Ámbito 264
Funciones anónimas 268
Funciones estándar 269
6.3 Objetos **270**
Declarando objetos 271
Métodos 273
La palabra clave this 274
Constructores 275
El operador new 278
Herencia 279
6.4 Objetos estándar **281**
Objetos String 283
Objetos Array 288
Objetos Date 295
Objeto Math 300
Objeto Window 302
Objeto Document 307
Objetos Element 312
Creando objetos Element 321
6.5 Eventos **322**
El método addEventListener() 323
Objetos Event 325
6.6 Depuración **335**
Consola 336
Objeto Console 337
Evento error 339
Excepciones 340
6.7 API **341**
Librerías nativas 342
Librerías externas 342

Capítulo 7—API Formularios
7.1 Procesando formularios **345**
7.2 Validación **348**
Errores personalizados 348
El evento invalid 350
El objeto ValidityState 351
7.3 Seudoclases **353**
Valid e Invalid 354
Optional y Required 354
In-range y Out-of-range 355

Capítulo 8—Medios
8.1 Vídeo **357**
Formatos de vídeo 360
8.2 Audio **361**
8.3 API Media **363**
Reproductor de vídeo 364
8.4 Subtítulos **370**
8.5 API TextTrack **374**
Leyendo pistas 375
Leyendo cues 376
Agregando pistas 378

Capítulo 9—API Stream
9.1 Capturando medios **381**
El objeto MediaStreamTrack 383

Capítulo 10—API Fullscreen
10.1 Aplicaciones modernas **387**
Pantalla completa 387
Estilos de pantalla completa 389

Capítulo 11—API Canvas
11.1 Gráficos **391**
El lienzo 391
El contexto 391
11.2 Dibujando **392**
Rectángulos 392
Colores 394
Gradientes 394
Trazados 395
Líneas 402
Texto 403
Sombras 405
Transformaciones 406
Estado 408
La propiedad GlobalCompositeOperation 409
11.3 Imágenes **410**
Patrones 413
Datos de imagen 414
Origen cruzado 416
Extrayendo datos 417
11.4 Animaciones **420**
Animaciones simples 420
Animaciones profesionales 422
11.5 Vídeo **425**
Aplicación de la vida real 427

Capítulo 12—WebGL
12.1 Lienzo en 3D **429**
12.2 Three.js **429**
Renderer 430
Escena 430
Cámara 431
Mallas 432
Figuras primitivas 433
Materiales 434
Implementación 437
Transformaciones 439
Luces 440
Texturas 442
Mapeado UV 444
Texturas de lienzo 446
Texturas de vídeo 447
Modelos 3D 449
Animaciones 3D 451

Capítulo 13—API Pointer Lock
13.1 Puntero personalizado **463**
Captura del ratón 463

Capítulo 14—API Web Storage
14.1 Sistemas de almacenamiento **471**
14.2 Session Storage **471**
Almacenando datos 472
Leyendo datos 474
Eliminando datos 475
14.3 Local Storage **477**
Evento storage 478

Capítulo 15—API IndexedDB
15.1 Datos estructurados **481**
Base de datos 481
Objetos y almacenes de objetos 482
Índices 483
Transacciones 484
15.2 Implementación **484**
Abriendo la base de datos 486
Definiendo índices 487
Agregando objetos 488
Leyendo objetos 489
15.3 Listando datos **490**
Cursores 490
Orden 492
15.4 Eliminando datos **493**
15.5 Buscando datos **494**

Capítulo 16—API File
16.1 Archivos 497
Cargando archivos 497
Leyendo archivos 498
Propiedades 500
Blobs 501
Eventos 504

Capítulo 17—API Drag and Drop
17.1 Arrastrar y soltar 507
Validación 512
Imagen miniatura 514
Archivos 516

Capítulo 18—API Geolocation
18.1 Ubicación geográfica 519
Obteniendo la ubicación 520
Supervisando la ubicación 523
Google Maps 524

Capítulo 19—API History
19.1 Historial 527
Navegación 527
URL 528
La propiedad state 530
Aplicación de la vida real 532

Capítulo 20—API Page Visibility
20.1 Visibilidad 535
Estado 535
Sistema de detección completo 537

Capítulo 21—Ajax Level 2
21.1 El Objeto XMLHttpRequest 539
Propiedades 542
Eventos 543
Enviando datos 544
Subiendo archivos 546
Aplicación de la vida real 549

Capítulo 22—API Web Messaging
22.1 Mensajería 553
Enviando un mensaje 553
Filtros y origen cruzado 556

Capítulo 23—API WebSocket
23.1 Web Sockets 559
Servidor WebSocket 559
Conectándose al servidor 561

Capítulo 24—API WebRTC

24.1 Paradigmas Web **567**
Servidores ICE 568
Conexión 569
Candidato ICE 569
Ofertas y respuestas 569
Descripción de la sesión 570
Transmisiones de medios 570
Eventos 571
24.2 Configuración **571**
Configurando el servidor de señalización 571
Configurando los servidores ICE 573
24.3 Implementando WebRTC **573**
24.4 Canales de datos **579**

Capítulo 25—API Web Audio

25.1 Estructura de audio **585**
Contexto de audio 586
Fuentes de audio 586
Conectando nodos 588
25.2 Aplicaciones de audio **588**
Bucles y tiempos 590
Nodos de audio 591
AudioParam 592
GainNode 593
DelayNode 594
BiquadFilterNode 596
DynamicsCompressorNode 596
ConvolverNode 597
PannerNode y sonido 3D 598
AnalyserNode 602

Capítulo 26—API Web Workers

26.1 Procesamiento paralelo **605**
Workers 605
Enviando y recibiendo mensajes 605
Errores 608
Finalizando workers 609
API síncronas 611
Importando código JavaScript 611
Workers compartidos 612

Índice **617**

Introducción

Internet se ha convertido en una parte esencial de nuestras vidas y la Web es la pieza central que conecta todas las tecnologías involucradas. Desde noticias y entretenimientos hasta aplicaciones móviles y videojuegos, todo gira en torno a la Web. Debemos acceder a un sitio web para abrir una cuenta por cada servicio que usamos, para conectar nuestras aplicaciones y dispositivos móviles entre sí, o para compartir la puntuación alcanzada en nuestro juego preferido. La Web es el centro de operaciones de nuestra actividad diaria, y HTML5 es lo que lo ha hecho posible.

Todo comenzó tiempo atrás con una versión simplificada de un lenguaje de programación llamado *HTML*. El lenguaje, junto con identificadores y protocolos de comunicación, se concibió con el propósito de ofrecer la base necesaria para la creación de la Web. El propósito inicial del HTML era estructurar texto para poder compartir documentos entre ordenadores remotos. Con el transcurso del tiempo, la introducción de mejores sistemas y pantallas de color obligaron al lenguaje a evolucionar y poder así trabajar con otros medios además de texto, como imágenes y tipos de letras personalizados. Esta expansión complicó el trabajo de los desarrolladores, a quienes les resultaba cada vez más difícil crear y mantener sitios web extensos usando solo HTML. El problema se resolvió con la incorporación de un nuevo lenguaje llamado CSS, el cual permite a los desarrolladores preparar el documento que se va a presentar en pantalla.

La asociación entre HTML y CSS simplificó el trabajo de los desarrolladores, pero la capacidad de estos lenguajes para responder al usuario o realizar tareas como la reproducción de vídeo o audio era aún muy limitada. Al principio, algunas compañías independientes ofrecieron sus propias alternativas. Los lenguajes de programación como Java y Flash se volvieron muy populares, pero resultaron incapaces de ofrecer una solución definitiva. Las herramientas producidas con estas tecnologías aún operaban desconectadas del contenido y solo compartían con el documento un espacio en la pantalla. Esta débil asociación allanó el camino para la evolución de un lenguaje que ya se encontraba incluido en los navegadores y que, por lo tanto, estaba fuertemente integrado en HTML. Este lenguaje, llamado *JavaScript*, permitía a los desarrolladores acceder al contenido del documento y modificarlo de forma dinámica, solicitar datos adicionales desde el servidor, procesar información y mostrar los resultados en la pantalla, convirtiendo los sitios web en pequeñas aplicaciones. Originalmente, el rendimiento de los navegadores no era lo suficientemente bueno como para realizar algunas de estas tareas, pero con la incorporación de mejores intérpretes, los desarrolladores encontraron formas de aprovechar las capacidades de este lenguaje y comenzaron a crear aplicaciones útiles, confirmando a JavaScript como la mejor opción para complementar HTML y CSS.

Con la combinación de HTML, CSS y JavaScript, las tecnologías requeridas para construir la Web de las que disfrutamos hoy en día estaban listas, pero todavía existía un problema que resolver. Estos lenguajes habían sido desarrollados de forma independiente y, por lo tanto, seguían sus propios caminos, ajenos a los cambios presentados por los demás. La solución surgió con la definición de una nueva especificación llamada *HTML5*. HTML5 unifica todas las tecnologías involucradas en el desarrollo web. A partir de ahora, HTML se encarga de definir la estructura del documento, CSS prepara esa estructura y su contenido para ser mostrado en pantalla, y JavaScript introduce la capacidad de procesamiento necesaria para construir aplicaciones web completamente funcionales.

La integración entre HTML, CSS y JavaScript bajo el amparo de HTML5 cambió la Web para siempre. De la noche a la mañana se crearon nuevas compañías basadas en aplicaciones web y mercados completos, lo que originó una era de oro para el desarrollo web.

Implementando estas tecnologías, las oportunidades son infinitas. La Web está aquí para quedarse y tú puedes ser parte de ella.

IMPORTANTE: en el momento de escribir este libro, la mayoría de los navegadores admite HTML5, pero algunos aún presentan limitaciones. Por este motivo, le recomendamos ejecutar los ejemplos del libro en las últimas versiones de Google Chrome y Mozilla Firefox (**www.google.com/chrome** y **www.mozilla.com**). Si lo necesita, puede consultar el estado de la implementación de estas tecnologías en **www.caniuse.com**. En la parte inferior de la primera página del libro encontrará el código de acceso que le permitirá acceder de forma gratuita a los contenidos adicionales del libro en **www.marcombo.info**.

Capítulo 1
Desarrollo web

1.1 Sitios web

Los sitios web son archivos que los usuarios descargan con sus navegadores desde ordenadores remotos. Cuando un usuario decide acceder a un sitio web, le comunica al navegador la dirección del sitio y el programa descarga los archivos, procesa su contenido y lo muestra en pantalla.

Debido a que los sitios webs son de acceso público e Internet es una red global, estos archivos deben estar siempre disponibles. Por este motivo, los sitios web no se almacenan en ordenadores personales, sino en ordenadores especializados diseñados para despachar estos archivos a los usuarios que los solicitan. El ordenador que almacena los archivos y datos de un sitio web se llama *servidor* y el ordenador que accede a esta información se llama *cliente*, tal como lo ilustra la Figura 1-1.

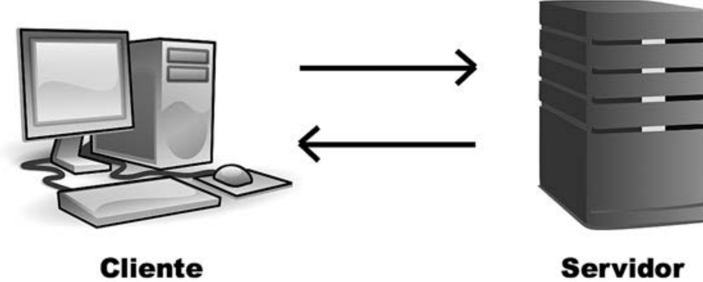

Figura 1-1: *Clientes y servidores*

Los servidores son muy similares a los ordenadores personales, con la diferencia de que están continuamente conectados a la red y ejecutando programas que les permiten responder a las solicitudes de los usuarios, sin importar cuándo se reciben o de donde proceden. Los programas más populares para servidores son Apache, para sistemas Linux, e IIS (Internet Information Server), creado por Microsoft para sistemas Windows. Entre otras funciones, estos programas son responsables de establecer la conexión entre el cliente y el servidor, controlar el acceso de los usuarios, administrar los archivos, y despachar los documentos y recursos requeridos por los clientes.

Archivos

Los sitios web están compuestos de múltiples documentos que el navegador descarga cuando el usuario los solicita. Los documentos que conforman un sitio web se llaman *páginas* y el proceso de abrir nuevas páginas *navegar* (el usuario navega a través de las páginas del sitio). Para desarrollar un sitio web, tenemos que crear un archivo por cada página que queremos incluir. Junto con estos archivos, también debemos incluir los archivos con las imágenes y cualquier otro recurso que queremos mostrar dentro de estas páginas (las imágenes y otros

medios gráficos se almacenan en archivos aparte). En la Figura 1-2 se representa cómo se muestran los directorios y archivos de un sitio web una vez que se suben al servidor.

..
imagenes
recursos
contacto.html
index.html
noticias.html

Figura 1-2: *Archivos de un sitio web*

En el ejemplo de la Figura 1-2 se incluyen dos directorios llamados *imagenes* y *recursos,* y tres archivos llamados *contacto.html*, *index.html* y *news.html*. Los directorios se crearon para almacenar las imágenes que queremos mostrar dentro de las páginas web y otros recursos, como los archivos que contienen los códigos en CSS y JavaScript. Por otro lado, los archivos de este ejemplo representan las tres páginas web que queremos incluir en este sitio. El archivo index.html contiene el código y la información correspondiente a la página principal (la página que el usuario ve cuando entra a nuestro sitio web por primera vez), el archivo contacto.html contiene el código necesario para presentar un formulario que el usuario puede rellenar para enviarnos un mensaje y el archivo noticias.html contiene el código necesario para mostrar las noticias que queremos compartir con nuestros usuarios. Cuando un usuario accede a nuestro sitio web por primera vez, el navegador descarga el archivo index.html y muestra su contenido en la ventana. Si el usuario realiza una acción para ver las noticias ofrecidas por nuestro sitio web, el navegador descarga el archivo noticias.html desde el servidor y reemplaza el contenido del archivo index.html por el contenido de este nuevo archivo. Cada vez que el usuario quiere acceder a una nueva página web, el navegador tiene que descargar el correspondiente archivo desde el servidor, procesarlo y mostrar su contenido en la pantalla.

Los archivos de un sitio web son iguales que los archivos que podemos encontrar en un ordenador personal. Todos tiene un nombre seleccionado por el desarrollador y una extensión que refleja el lenguaje usado para programar su contenido (en nuestro ejemplo, los archivos tienen la extensión .html porque fueron programados en HTML). Aunque podemos asignar cualquier nombre que queramos a estos archivos, el archivo que genera la página inicial presenta algunos requisitos. Algunos servidores como Apache designan archivos por defecto en caso de que el usuario no especifique ninguno. El nombre utilizado con más frecuencia es *index*. Si un usuario accede al servidor sin especificar el nombre del archivo que intenta abrir, el servidor busca un archivo con el nombre *index* y lo envía de vuelta al cliente. Por esta razón, el archivo index es el punto de entrada de nuestro sitio web y siempre debemos incluirlo.

IMPORTANTE: los servidores son flexibles en cuanto a los nombres que podemos asignar a nuestros archivos, pero existen algunas reglas que debería seguir para asegurarse de que sus archivos son accesibles. Evite usar espacios. Si necesita separar palabras use el guion bajo en su lugar (_). Además, debe considerar que algunos caracteres realizan funciones específicas en la Web, por lo que es mejor evitar caracteres especiales como ?, %, #, /, y usar solo letras minúsculas sin acentos y números.

Lo básico: aunque *index* es el nombre más común, no es el único que podemos asignar al archivo por defecto. Algunos servidores designan otros nombres como *home* o *default*, e incluyen diferentes extensiones. Por ejemplo, si en lugar de programar nuestros documentos en HTML lo hacemos en un lenguaje de servidor como PHP, debemos asignar a nuestro archivo index el nombre *index.php*. El servidor contiene una lista de archivos y continúa buscando hasta que encuentra uno que coincida con esa lista. Por ejemplo, Apache primero busca por un archivo con el nombre index y la extensión .html, pero si no lo encuentra, busca por un archivo con el nombre index y la extensión .php. Estudiaremos HTML y PHP más adelante en este y otros capítulos.

Dominios y URL

Los servidores se identifican con un valor llamado *IP* (Internet Protocol). Esta IP es única para cada ordenador y, por lo tanto, trabaja como una dirección que permite ubicar a un ordenador dentro de una red. Cuando el navegador tiene que acceder a un servidor para descargar el documento solicitado por el usuario, primero busca el servidor a través de esta dirección IP y luego le pide que le envíe el documento.

Las direcciones IP están compuestas por números enteros entre 0 y 255 separados por un punto, o números y letras separadas por dos puntos, dependiendo de la versión (IPv4 o IPv6). Por ejemplo, la dirección 216.58.198.100 corresponde al servidor donde se encuentra alojado el sitio web de Google. Si escribimos esta dirección IP en la barra de navegación de nuestro navegador, la página inicial de Google se descarga y muestra en pantalla.

En teoría, podríamos acceder a cualquier servidor utilizando su dirección IP, pero estos valores son crípticos y difíciles de recordar. Por esta razón, Internet utiliza un sistema que identifica a cada servidor con un nombre específico. Estos nombres personalizados, llamados *dominios*, son identificadores sencillos que cualquier persona puede recordar, como *google* o *yahoo*, con una extensión que determina el propósito del sitio web al que hacen referencia, como *.com* (comercial) o *.org* (organización). Cuando el usuario le pide al navegador que acceda al sitio web con el dominio www.google.com, el navegador accede primero a un servidor llamado *DNS* que contiene una lista de dominios con sus respectivas direcciones IP. Este servidor encuentra la IP 216.58.198.100 asociada al dominio www.google.com, la devuelve al navegador, y entonces el navegador accede al sitio web de Google por medio de esta IP. Debido a que las direcciones IP de los sitios web siempre se encuentran asociadas a sus dominios, no necesitamos recordar la dirección de un servidor para acceder a él, solo tenemos que recordar el domino y el navegador se encarga de encontrar el servidor y descargar los archivos por nosotros.

Los sitios web están compuestos por múltiples archivos, por lo que debemos agregar el nombre del archivo al dominio para indicar cuál queremos descargar. Esta construcción se llama *URL* e incluye tres partes, tal como se describe en la Figura 1-3.

Figura 1-3: *URL*

La primera parte de la URL es una cadena de caracteres que representa el protocolo de comunicación que se utilizará para acceder al recurso (el protocolo creado para la Web se llama *HTTP*), el siguiente componente es el dominio del sitio web y el último componente es el nombre del recurso que queremos descargar (puede ser un archivo, como en nuestro ejemplo, o una ruta a seguir que incluye el directorio donde el archivo se encuentra almacenado (por ejemplo, http://www.ejemplo.com/imagenes/milogo.jpg). La URL en nuestro ejemplo le pide al navegador que utilice el protocolo HTTP para acceder al archivo contacto.html, ubicado en el servidor identificado con el domino www.ejemplo.com.

Las URL se utilizan para ubicar cada uno de los documentos en el sitio web y son, por lo tanto, necesarias para navegar por el sitio. Si el usuario no especifica ningún archivo, el servidor devuelve el archivo por defecto, pero de ahí en adelante, cada vez que el usuario realiza una acción para abrir una página diferente, el navegador debe incluir en la URL el nombre del archivo que corresponde a la página solicitada.

IMPORTANTE: una vez que ha conseguido el dominio para su sitio web, puede crear subdominios. Los subdominios son enlaces directos a directorios y nos permiten crear múltiples sitios web en una misma cuenta. Un subdominio se crea con el nombre del directorio y el dominio conectados por un punto. Por ejemplo, si su dominio es www.ejemplo.com y luego crea un subdominio para un directorio llamado *recursos*, podrá acceder directamente al directorio escribiendo en el navegador la URL http://recursos.ejemplo.com.

Lo básico: existen diferentes protocolos que los ordenadores utilizan para comunicarse entre ellos, y transferir recursos y datos. HTTP (HyperText Transfer Protocol) es el protocolo de comunicación que se utiliza para acceder a documentos web. Siempre tenemos que incluir el prefijo HTTP en la URL cuando el recurso al que estamos tratando de acceder pertenece a un sitio web, pero en la práctica esto no es necesario porque los navegadores lo hacen de forma automática. Existe otra versión disponible de este protocolo llamado *HTTPS*. La S indica que la conexión es encriptada por protocolos de encriptación como TLS o SSL. Los sitios web pequeños no necesitan encriptación, pero se recomienda utilizarla en sitios web que manejan información sensible.

Hipervínculos

En teoría, podemos acceder a todos los documentos de un sitio web escribiendo la URL en la barra de navegación del navegador. Por ejemplo, si queremos acceder a la página inicial en español del sitio web *For Masterminds*, podemos insertar la URL http://www.formasterminds.com/esindex.php, o bien insertar la URL http://www.formasterminds.com/escontact.php para abrir la página que nos permite enviar un mensaje a su desarrollador. Aunque es posible acceder a todos los archivos del sitio web usando este método, no es práctico. En primer lugar, los usuarios no conocen los nombres que el desarrollador eligió para cada archivo y, por lo tanto, estarán limitados a aquellos nombres que pueden adivinar o solo a la página principal que devuelve por defecto. En segundo lugar, los sitios web pueden estar compuestos por docenas o incluso miles de páginas web (algunos

sitios contienen millones) y la mayoría de los documentos serían imposibles de encontrar. La solución se encontró con la definición de hipervínculos. Los hipervínculos, también llamados *enlaces*, son referencias a documentos dentro de las páginas de un sitio web. Incorporando estos enlaces, una página puede contener referencias a otras páginas. Si el usuario hace clic con el ratón en un enlace, el navegador sigue esa referencia y el documento indicado por la URL de la referencia se descarga y muestra en pantalla. Debido a estas conexiones entre páginas, los usuarios pueden navegar en el sitio web y acceder a todos sus documentos simplemente haciendo clic en sus enlaces.

Lo básico: los enlaces son lo que transforma a un grupo de archivos en un sitio web. Para crear un sitio web, debe programar los documentos correspondientes a cada página e incluir dentro de las mismas los enlaces que establecen una ruta que el usuario puede seguir para acceder a cada una de ellas. Estudiaremos cómo incorporar enlaces en nuestros documentos en el Capítulo 2.

URL absolutas y relativas

Los hipervínculos son procesados por el navegador antes de ser usados para acceder a los documentos. Por esta razón, se pueden definir con URL absolutas o relativas. Las URL absolutas son aquellas que incluyen toda la información necesaria para acceder al recurso (ver Figura 1-3), mientras que las relativas son aquellas que solo declaran la parte de la ruta que el navegador tiene que agregar a la URL actual para acceder al recurso. Por ejemplo, si tenemos un hipervínculo dentro de un documento que referencia una imagen dentro del directorio imagenes, podemos crear el enlace con la URL http://www.ejemplo.com/imagenes/miimagen.png, pero también tenemos la opción de declararla como "imagenes/miimagen.png" y el navegador se encargará de agregar a esta ruta la URL actual y descargar la imagen.

Las URL relativas no solo pueden determinar una ruta hacia abajo, sino también hacia arriba de la jerarquía. Por ejemplo, si tenemos un documento dentro del directorio recursos en el ejemplo de la Figura 1-2 y queremos acceder a un documento en el directorio raíz, podemos crear una URL relativa usando los caracteres ../ al comienzo de la ruta. Si el documento que queremos acceder es noticias.html, la URL relativa sería ../noticias.html. Los dos puntos .. le indican al navegador que el documento al que queremos acceder se encuentra dentro del directorio padre del actual directorio (recursos, en nuestro ejemplo).

1.2 Lenguajes

Como mencionamos en la introducción, HTML5 incorpora tres características (estructura, estilo, y funcionalidad), con lo que integra tres lenguajes de programación independientes: HTML, CSS, y JavaScript. Estos lenguajes están compuestos por grupos de instrucciones que los navegadores pueden interpretar para procesar y mostrar los documentos al usuario. Para crear nuestros documentos, tenemos que aprender todas las instrucciones incluidas en estos lenguajes y saber cómo organizarlas.

HTML

HTML (*HyperText Markup Language*) es un lenguaje compuesto por un grupo de etiquetas definidas con un nombre rodeado de paréntesis angulares. Los paréntesis angulares delimitan la etiqueta y el nombre define el tipo de contenido que representa. Por ejemplo, la etiqueta **`<html>`** indica que el contenido es código HTML. Algunas de estas etiquetas son declaradas individualmente (por ejemplo, **`<br>`**) y otras son declaradas en pares, que incluyen una de apertura y otra de cierre, como **`<html></html>`** (en la etiqueta de cierre el nombre va precedido por una barra invertida). Las etiquetas individuales y las de apertura pueden incluir atributos para ofrecer información adicional acerca de sus contenidos (por ejemplo, **`<html lang="es">`**). Las etiquetas individuales y la combinación de etiquetas de apertura y cierre se llaman *elementos*. Los elementos compuestos por una sola etiqueta se usan para modificar el contenido que los rodea o incluir recursos externos, mientras que los elementos que incluyen etiquetas de apertura y cierre se utilizan para delimitar el contenido del documento, tal como ilustra la Figura 1-4.

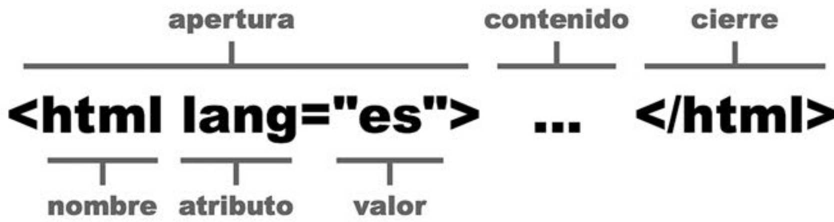

Figura 1-4: *Elemento HTML*

Se deben combinar múltiples elementos para definir un documento. Los elementos son listados en secuencia de arriba abajo y pueden contener otros elementos en su interior. Por ejemplo, el elemento **`<html>`** que se muestra en la Figura 1-4 declara que su contenido debe ser interpretado como código HTML. Por lo tanto, el resto de los elementos que describen el contenido de ese documento se deben declarar entre las etiquetas **`<html>`** y **`</html>`**. A su vez, los elementos dentro del elemento **`<html>`** pueden incluir otros elementos. El siguiente ejemplo muestra un documento HTML sencillo que incluye todos los elementos necesarios para definir una estructura básica y mostrar el mensaje HOLA MUNDO! en la pantalla.

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <title>Mi primer documento HTML</title>
  </head>
  <body>
    <p>HOLA MUNDO!</p>
  </body>
</html>
```

Listado 1-1: *Creando un documento HTML*

En el ejemplo del Listado 1-1 presentamos un código sencillo, pero con una estructura compleja. En la primera línea, se encuentra una etiqueta individual que declara el tipo de documento (**<!DOCTYPE html>**) seguida por una etiqueta de apertura **<html lang="es">**. Entre las etiquetas **<html>** y **</html>** se incluyen otros elementos que representan la cabecera y el cuerpo del documento (**<head>** y **<body>**), los cuales a su vez encierran más elementos con sus respectivos contenidos (**<title>** y **<p>**), demostrando cómo se compone un documento HTML. Los elementos se listan uno a continuación de otro y también dentro de otros elementos, de modo que se construye una estructura de tipo árbol con el elemento **<html>** como raíz.

Lo básico: en general, todo elemento puede ser anidado, convertirse en un contenedor o ser contenido por otros elementos. Los elementos exclusivamente estructurales como **<html>**, **<head>** y **<body>** tienen un lugar específico en un documento HTML, pero el resto son flexibles, tal como veremos en el Capítulo 2.

Como ya mencionamos, las etiquetas individuales y de apertura pueden incluir atributos. Por ejemplo, la etiqueta de apertura **<html>** declarada en el Listado 1-1 no está solo compuesta por el nombre **html** y los paréntesis angulares, sino también por el texto **lang="es"**. Este es un atributo con un valor. El nombre del atributo es **lang** y el valor **es** se asigna al atributo usando el carácter **=**. Los atributos ofrecen información adicional acerca del elemento y su contenido. En este caso, el atributo **lang** declara el idioma del contenido del documento (**es** por Español).

Lo básico: los atributos se declaran siempre dentro de la etiqueta de apertura (o etiquetas individuales) y pueden tener una estructura que incluye un nombre y un valor, como el atributo **lang** de la etiqueta **<html>**, o representar un valor por sí mismos, como el atributo **html** de la etiqueta **<!DOCTYPE>**. Estudiaremos los elementos HTML y sus atributos en el Capítulo 2.

CSS

CSS (*Cascading Style Sheets*) es el lenguaje que se utiliza para definir los estilos de los elementos HTML, como el tamaño, el color, el fondo, el borde, etc. Aunque todos los navegadores asignan estilos por defecto a la mayoría de los elementos, estos estilos generalmente están lejos de lo que queremos para nuestros sitios web. Para declarar estilos personalizados, CSS utiliza propiedades y valores. Esta construcción se llama *declaración* y su sintaxis incluye dos puntos después del nombre de la propiedad, y un punto y coma al final para cerrar la línea.

Figura 1-5: *Propiedad CSS*

En el ejemplo de la Figura 1-5, el valor **#FF0000** se asigna a la propiedad **color**. Si esta propiedad se aplica luego a un elemento HTML, el contenido de ese elemento se mostrará en color rojo (el valor **#FF0000** representa el color rojo).

Las propiedades CSS se pueden agrupar usando llaves. Un grupo de una o más propiedades se llama *regla* y se identifica por un nombre llamado *selector*.

```
body {
  width: 100%;
  margin: 0px;
  background-color: #FF0000;
}
```

Listado 1-2: *Declarando reglas CSS*

El Listado 1-2 declara una regla con tres propiedades: **`width`**, **`margin`** y **`background-color`**. Esta regla se identifica con el nombre **`body`**, lo que significa que las propiedades serán aplicadas al elemento **`<body>`**. Si incluimos esta regla en un documento, el contenido del documento se extenderán hacia los límites de la ventana del navegador y tendrán un fondo rojo.

Lo básico: existen diferentes técnicas para aplicar estilos CSS a elementos HTML. Estudiaremos las propiedades CSS y cómo incluirlas en un documento HTML en los Capítulos 3 y 4.

JavaScript

A diferencia de HTML y CSS, JavaScript es un lenguaje de programación. Para ser justos, todos estos lenguajes pueden ser considerados lenguajes de programación, pero en la práctica existen algunas diferencias en la forma en la que suministran las instrucciones al navegador. HTML es como un grupo de indicadores que el navegador interpreta para organizar la información, CSS puede ser considerado como una lista de estilos que ayudan al navegador a preparar el documento para ser presentado en pantalla (aunque la última especificación lo convirtió en un lenguaje más dinámico), pero JavaScript es un lenguaje de programación, comparable con cualquier otro lenguaje de programación profesional como C++ o Java. JavaScript difiere de los demás lenguajes en que puede realizar tareas personalizadas, desde almacenar valores hasta calcular algoritmos complejos, incluida la capacidad de interactuar con los elementos del documento y procesar su contenido dinámicamente.

Al igual que HTML y CSS, JavaScript se incluye en los navegadores y, por lo tanto, se encuentra disponible en todos nuestros documentos. Para declarar código JavaScript dentro de un documento, HTML ofrece el elemento **`<script>`**. El siguiente ejemplo es una muestra de un código escrito en JavaScript.

```
<script>
  function cambiarColor() {
    document.body.style.backgroundColor = "#0000FF";
  }
  document.addEventListener("click", cambiarColor);
</script>
```

Listado 1-3: *Declarando código JavaScript*

El código en el Listado 1-3 cambia el color de fondo del elemento **<body>** a azul cuando el usuario hace clic en el documento.

Lo básico: con el elemento **<script>** también podemos cargar código JavaScript desde archivos externos. Estudiaremos el elemento **<script>** en el Capítulo 2 y el lenguaje JavaScript en el Capítulo 6.

Lenguajes de servidor

Los códigos programados en HTML, CSS, y JavaScript son ejecutados por el navegador en el ordenador del usuario (el cliente). Esto significa que, después de que los archivos del sitio web se suben al servidor, permanecen inalterables hasta que se descargan en un ordenador personal y sus códigos son ejecutados por el navegador. Aunque esto permite la creación de sitios web útiles e interactivos, hay momentos en los cuales necesitamos procesar la información en el servidor antes de enviarla al usuario. El contenido producido por esta información se denomina *contenido dinámico*, y es generado por códigos ejecutados en el servidor y programados en lenguajes que fueron especialmente diseñados con este propósito (lenguajes de servidor). Cuando el navegador solicita un archivo que contiene este tipo de código, el servidor lo ejecuta y luego envía el resultado como respuesta al usuario. Estos códigos no solo se utilizan para generar contenido y documentos en tiempo real, sino también para procesar la información enviada por el navegador, almacenar datos del usuario en el servidor, controlar cuentas, etc.

Existen varios lenguajes disponibles para crear código ejecutable en los servidores. Los más populares son PHP, Ruby, y Python. El siguiente ejemplo es una muestra de un código escrito en PHP.

```
<?php
  $nombre = $_GET['minombre'];
  print('Su nombre es: '.$nombre);
?>
```

Listado 1-4: *Declarando código ejecutable en el servidor*

El código del Listado 1-4 recibe un valor enviado por el navegador, lo almacena en la memoria y crea un mensaje con el mismo. Cuando se ejecuta este código, se crea un nuevo documento que contiene el mensaje final, el archivo se envía de vuelta al cliente y finalmente el navegador muestra su contenido en pantalla.

IMPORTANTE: los lenguajes de servidor utilizan su propia tecnología, pero trabajan junto con HTML5 para llevar un registro de las cuentas de usuarios, almacenar información en el servidor, manejar bases de datos, etc. El tema va más allá del propósito de este libro. Para obtener más información sobre cómo programar en PHP, Ruby, o Python, descargue los contenidos adiconales en **www.marcombo.info**.

1.3 Herramientas

Crear un sitio web involucra múltiples pasos. Tenemos que programar los documentos en HTML, crear los archivos con los estilos CSS y los códigos en JavaScript, configurar el servidor

que hará que el sitio sea visible a los usuarios y transferir todos los archivos desde nuestro ordenador al servidor. Por fortuna existen muchas herramientas disponibles que nos pueden ayudar con estas tareas. Estas herramientas son muy fáciles de usar y la mayoría se ofrecen de forma gratuita.

IMPORTANTE: en esta sección del capítulo introducimos todas las herramientas que necesitará para crear sus sitios web y ofrecerlos a sus usuarios. Esto incluye las herramientas requeridas para programar y diseñar un sitio web, pero también otras que necesitará para configurarlo y probarlo antes de hacerlo público. La mayoría de los ejemplos de este libro no tienen que subirse a un servidor para poder trabajar adecuadamente y, por lo tanto, puede ignorar parte de esta información hasta que sea requerida por sus proyectos.

Editores

Los documentos HTML, así como los archivos CSS y JavaScript, son archivos de texto, por lo que podemos usar cualquier editor incluido en nuestro ordenador para crearlos, como el bloc de notas de Windows o la aplicación editor de texto de los ordenadores de Apple, pero también existen editores de texto especialmente diseñados para programadores y desarrolladores web que pueden simplificar nuestro trabajo. Estos editores resaltan texto con diferentes colores para ayudarnos a identificar cada parte del código, o listan los archivos de un proyecto en un panel lateral para ayudarnos a trabajar con múltiples archivos al mismo tiempo. La siguiente es una lista de los editores y de IDE (Integrated Development Environments) más populares disponibles para ordenadores personales y ordenadores de Apple.

- **Atom** (www.atom.io) es un editor gratuito, simple de usar y altamente personalizable (recomendado).
- **Brackets** (www.brackets.io) es un editor gratuito creado por Adobe.
- **KompoZer** (www.kompozer.net) es un editor gratuito con un panel de vista previa que facilita la búsqueda de partes específicas del documento a modificar.
- **Aptana** (www.aptana.com) es una IDE gratuita con herramientas que simplifican la administración de archivos y proyectos.
- **NetBeans** (www.netbeans.org) es una IDE gratuita con herramientas para administrar archivos y proyectos.
- **Sublime** (www.sublimetext.com) es un editor de pago con una versión gratuita de evaluación.
- **Komodo** (www.komodoide.com) es una IDE de pago que puede trabajar con una cantidad extensa de lenguajes de programación.
- **Dreamweaver** (www.adobe.com/products/dreamweaver.html) es una IDE de pago con tecnología WYSIWYG incorporada (Lo Que Ves Es Lo Que Obtienes) que nos permite ver los resultados de la ejecución del código en tiempo real.

Trabajar con un editor es simple: tenemos que crear un directorio en nuestro disco duro donde vamos a almacenar los archivos del sitio web, abrir el editor, y crear dentro de este directorio todos los archivos y directorios adicionales que necesitamos para nuestro proyecto. La Figura 1-6 muestra cómo se ve el editor Atom cuando se abre por primera vez.

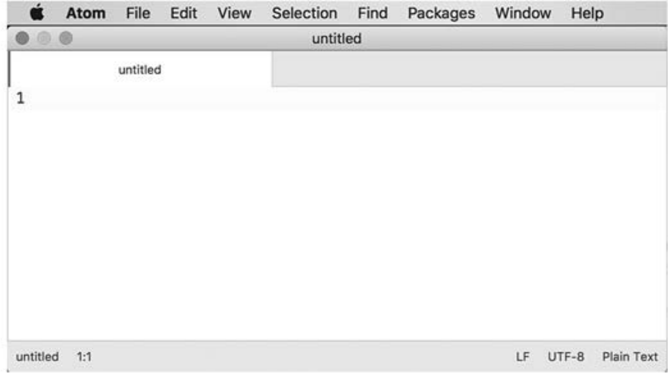

Figura 1-6: *Editor Atom con un archivo vacío*

Este editor tiene una opción en el menú **File** (Archivo) para abrir un proyecto (**Add Project Folder**). La opción se muestra en la Figura 1-7, número 1.

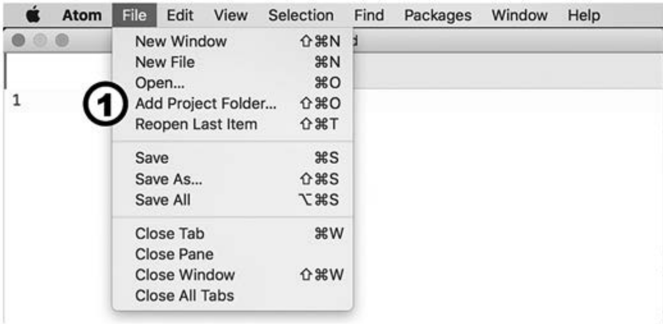

Figura 1-7: *Opción para agregar un proyecto*

Si hacemos clic en esta opción y luego seleccionamos el directorio creado para nuestro proyecto, el editor abre un nuevo panel a la izquierda con la lista de archivos dentro del directorio. La Figura 1-8, a continuación, muestra un directorio llamado **Test** creado para contener los archivos del ejemplo de la Figura 1-2.

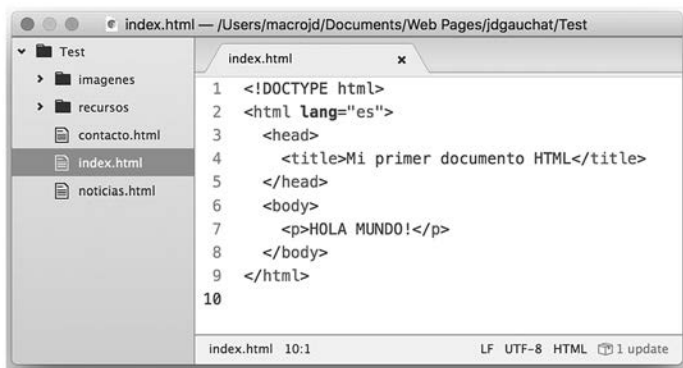

Figura 1-8: *Archivos del proyecto*

Hágalo usted mismo: visite **www.atom.io** para descargar el editor Atom. Una vez que el editor esté instalado en su ordenador, cree un nuevo directorio en su disco duro para almacenar los archivos de su sitio web. Abra Atom, vaya al menú **File** y seleccione la opción **Add Project Folder** (Figura 1-7, número 1). Seleccione el directorio que acaba de crear. Abra de nuevo el menú **File** y seleccione la opción **New File** (Nuevo Archivo). Copie el código del Listado 1-1 y grabe el archivo con el nombre index.html. Después de seleccionar el archivo en el panel de la izquierda, debería ver algo similar a lo que se muestra en la Figura 1-8.

IMPORTANTE: explicar cómo trabajar con Atom, u otro editor, va más allá del propósito de este libro, pero hemos incluido enlaces a cursos y vídeos en nuestro sitio web con más información. Para acceder a ellos, descargue los contenidos adicionales y haga clic en las opciones **Enlaces** y **Vídeos**.

Registro de dominios

Una vez que nuestro sitio web está listo para ser presentado en público, tenemos que registrar el dominio que los usuarios van a escribir en la barra de navegación para acceder a él. Como ya mencionamos, un domino es simplemente un nombre personalizado con una extensión que determina el propósito del sitio web. El nombre puede ser cualquiera que deseemos, y contamos con varias opciones para definir la extensión, desde extensiones con propósitos comerciales, como .com o .biz., a aquellas sin ánimo de lucro o personales, como .org, .net o .info, por no mencionar las extensiones regionales que incluyen un valor adicional para determinar la ubicación del sitio web, como .co.uk para sitios web en el Reino Unido o .eu para sitios web relacionados con la Union Europea.

Para obtener un domino para nuestro sitio web, tenemos que abrir una cuenta con un registrante y adquirirlo. La mayoría de los dominios requieren del pago de un arancel anual, pero el proceso es relativamente sencillo y hay muchas compañías disponibles que pueden hacerse cargo del trámite por nosotros. La más popular es GoDaddy (www.godaddy.com), pero la mayoría de las compañías que ofrecen servicios para desarrolladores también incluyen la posibilidad de registrar un dominio. Como dijimos, el proceso de registro es sencillo; tenemos que decidir el nombre y la extensión que vamos a asignar a nuestro dominio, realizar una búsqueda para asegurarnos de que el nombre que hemos elegido no está siendo utilizado y se encuentra disponible, y luego hacer el pedido (las compañías mencionadas con anterioridad facilitan todas las herramientas necesarias para este propósito).

Cuando el dominio está registrado, el sistema nos pide los nombres de servidores (nameservers) que queremos asociar al dominio. Estos nombres son cadenas de texto compuestas por un dominio y un prefijo, generalmente NS1 y NS2, que determinan la ubicación de nuestro sitio web (los nombres de servidor o nameservers los facilita el servidor en el que se almacena nuestro sitio web). Si aún no contamos con estos nombres, podemos usar los que ofrece la compañía y cambiarlos más adelante.

IMPORTANTE: la compañía que registra su dominio asigna nombres de servidor por defecto que ellos usan como destino provisional (también conocido como *aparcamiento* o *parking*). En principio puede asignar estos nombres y cambiarlos más adelante cuando su servidor esté listo. Algunas compañías ofrecen el registro de dominio junto con el alquiler de servidores y, por lo tanto, pueden encargarse de la configuración del dominio por nosotros si usamos sus servidores.

Alojamiento web

Configurar y mantener un servidor exige conocimientos que no todos los desarrolladores poseen. Por este motivo, existen compañías que ofrecen un servicio llamado alojamiento web (*web hosting*), que permite a cualquier individuo alquilar un servidor configurado y listo para almacenar, y operar uno o múltiples sitios web.

Existen diferentes tipos de alojamiento web disponible, desde aquellos que permiten que varios sitios web operen desde un mismo servidor (alojamiento compartido) hasta servicios más profesionales que reservan un servidor completo para un único sitio web (alojamiento dedicado), o distribuyen un sitio web extenso en muchos servidores (alojamiento en la nube), incluidas varias opciones intermedias.

La principal ventaja de tener una cuenta de alojamiento web es que todas ofrecen un panel de control con opciones para crear y configurar nuestro sitio web. Las siguientes son las opciones más comunes que nos encontraremos en la mayoría de estos servicios.

- **File Manager** es una herramienta web que nos permite administrar los archivos de nuestro sitio. Con esta herramienta podemos subir, bajar, editar o eliminar archivos en el servidor desde el navegador, sin tener que usar ninguna otra aplicación.
- **FTP Accounts** es un servicio que nos permite administrar las cuentas que usamos para conectarnos al servidor por medio de FTP. FTP (File Transfer Protocol) es un protocolo de comunicación diseñado para transferir archivos desde un ordenador a otro en la red.
- **MySQL Databases** es un servicio que nos permite crear bases de datos para nuestro sitio web.
- **phpMyAdmin** es una aplicación programada en PHP que podemos usar para administrar las bases de datos creadas para nuestro sitio web.
- **Email Accounts** es un servicio que nos permite crear cuentas de email con el dominio de nuestro sitio web (por ejemplo, info@midominio.com).

El panel de control más popular en el mercado es *cPanel*. La Figura 1-9 muestra el diseño y algunas de las opciones que ofrece.

Figura 1-9: *Opciones ofrecidas por cPanel*

El coste de una cuenta de alojamiento puede variar entre algunos dólares por una cuenta compartida hasta cientos de dólares al mes por un servidor dedicado. Una vez que abrimos la cuenta, la compañía nos envía un email con la información que necesitamos para acceder al panel de control y configurar el servidor. El sistema de la compañía generalmente crea todas las cuentas básicas que necesitamos, incluida una cuenta FTP para subir los archivos, tal como veremos a continuación.

Lo básico: además de las cuentas de alojamiento de pago, existe el alojamiento gratuito que podemos usar para practicar, pero estos servicios incluyen propaganda o imponen restricciones que impiden el desarrollo de sitios web profesionales. Siempre se recomienda comenzar con una cuenta de alojamiento compartido que puede costar unos 5 dólares al mes para aprender cómo trabaja un servicio de alojamiento profesional y estudiar todas las opciones que ofrece. Varias compañías incluyen en sus servicios este tipo de alojamiento. Las más populares en este momento son www.godaddy.com y www.hostgator.com.

Programas FTP

Como acabamos de mencionar, las cuentas de alojamiento web ofrecen un servicio para administrar los archivos del sitio web desde el navegador. Esta es una página web a la que podemos acceder desde el panel de control para subir, bajar y editar los archivos en el servidor. Es una herramienta útil, pero solo práctica cuando necesitamos realizar pequeñas modificaciones o subir unos pocos archivos. La herramienta aprovecha un sistema que se encuentra integrado en los navegadores y que trabaja con un protocolo llamado *FTP* (File Transfer Protocol) usado para transferir archivos desde un ordenador a otro en una red. Los navegadores incluyen este sistema porque lo necesitan para permitir a los usuarios descargar archivos pero, debido a que su principal propósito es descargar y mostrar sitios web en la pantalla, ofrecen una mala experiencia a la hora de manipular estos archivos. Por esta razón, los desarrolladores profesionales no utilizan el navegador sino programas diseñados específicamente para transferir archivos entre un cliente y un servidor usando el protocolo FTP.

El mercado ofrece varios programas FTP, incluidas versiones de pago y gratuitas. El programa gratuito más popular se llama *Filezilla* y se encuentra disponible en **www.filezilla-project.org**. Este programa ofrece varios paneles con información acerca de la conexión y los ordenadores que participan, incluidos dos paneles lado a lado con la lista de los archivos locales y remotos que podemos transferir entre ordenadores con solo arrastrarlos de un panel a otro.

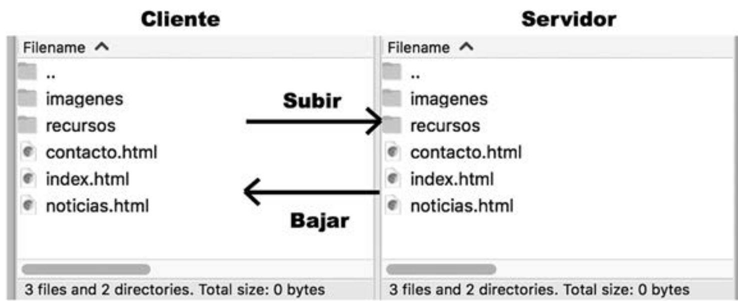

Figura 1-10: *Interfaz de Filezilla*

Cuando abrimos una cuenta de alojamiento, el sistema crea automáticamente una cuenta FTP para nuestro sitio web que incluye el nombre de usuario y clave requeridos para conectarse al servidor usando este protocolo (si el sistema no configura esta cuenta, podemos hacerlo nosotros mismos desde la opción **FTP Accounts** en el panel de control). Los valores que necesitamos para realizar la conexión son el host (IP o dominio), el usuario y la clave, además del puerto asignado por el servidor para conectarse por medio de FTP (por defecto, 21). Filezilla ofrece dos maneras de insertar esta información: una barra en la parte superior con la que realizar una conexión rápida (Figura 1-11, número 2) y un botón para almacenar múltiples conexiones de uso frecuente (Figura 1-11, número 1).

Figura 1-11: *Configuración de conexiones*

Si pulsamos el botón para almacenar o acceder a conexiones previas (número 1), Filezilla abre una ventana donde podemos administrar la lista de conexiones disponibles y especificar opciones adicionales de configuración. La ventana incluye botones para crear, renombrar y borrar una conexión (**New Site**, **Rename**, **Delete**), campos donde podemos seleccionar el protocolo que deseamos utilizar (FTP para una conexión normal y SFTP para una conexión segura), el modo de encriptación usado para transferir los archivos y el tipo de cuenta requerida (**Anonymous** para conexiones anónimas o **Normal** para conexiones que requieren usuario y clave). El programa también incluye paneles adicionales para una configuración más avanzada, tal como muestra la Figura 1-12.

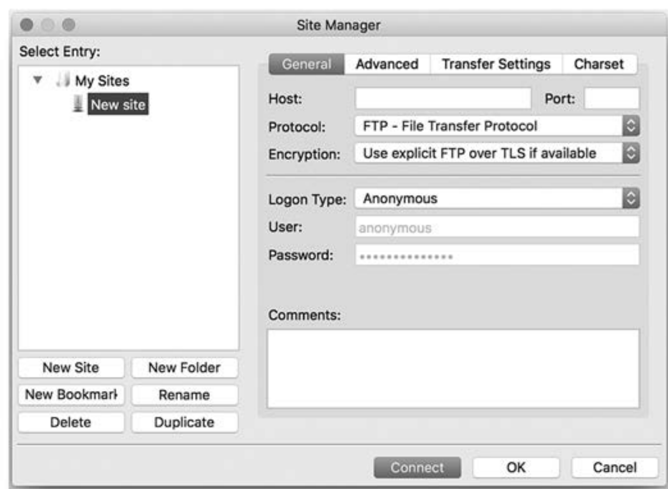

Figura 1-12: *Administrador de conexiones*

En una situación normal, para establecer la conexión tenemos que insertar el host (el IP o dominio de nuestro sitio web), seleccionar el tipo de cuenta como **Normal**, introducir el nombre de usuario y la contraseña, y dejar el resto de los valores por defecto. Una vez que los ordenadores se conectan, Filezilla muestra la lista de archivos en la pantalla. A la izquierda se

encuentran los archivos en el directorio seleccionado en nuestro ordenador (podemos seleccionar cualquier directorio que queramos en nuestro disco duro), y a la derecha se encuentran los archivos y directorios disponibles en el directorio raíz de nuestra cuenta de alojamiento, como muestra la Figura 1-13.

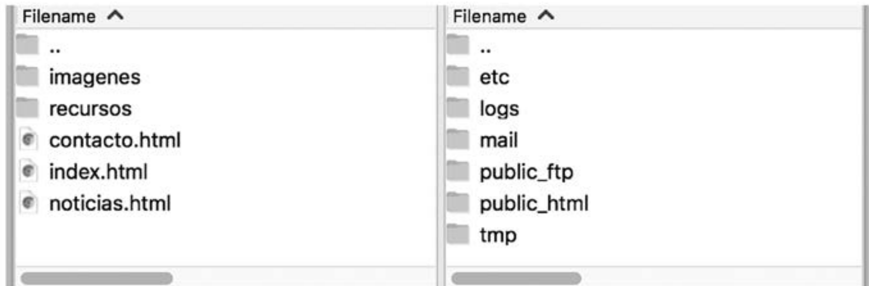

Figura 1-13: *Contenido del directorio raíz*

Cuando se crea una cuenta de alojamiento, el sistema incluye varios directorios y archivos para almacenar la información requerida por el servicio (almacenar emails, hacer un seguimiento de la actividad de los usuarios, etc.). El directorio en el que se deben almacenar los archivos de nuestro sitio web se llama *public_html*. Una vez que se abre este directorio, podemos comenzar a subir nuestros archivos arrastrándolos desde el panel de la izquierda al panel de la derecha (ver Figura 1-10).

IMPORTANTE: va más allá del propósito de este libro explicar cómo funciona Filezilla o cualquier otro programa de FTP, pero hemos incluido enlaces a cursos y vídeos en nuestro sitio web con más información. Para acceder a ellos, descargue los contenidos adicionales y haga clic en las opciones **Enlaces** y **Vídeos**.

MAMP

Los documentos HTML se pueden abrir directamente en un ordenador personal. Por ejemplo, si abrimos el archivo index.html con el documento creado en el Listado 1-1, la ventana del navegador muestra el texto HOLA MUNDO!, según ilustra la Figura 1-14 debajo (el resto de los elementos de este documento son estructurales y, por lo tanto, no producen resultados visibles).

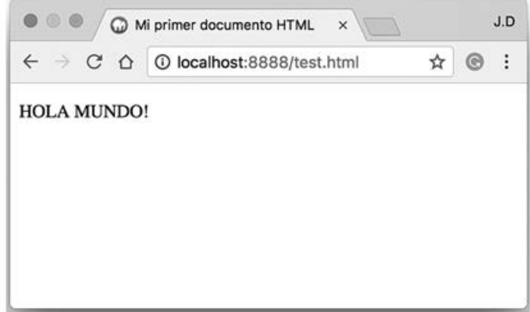

Figura 1-14: *Documento HTML en el navegador*

Lo básico: para abrir un archivo en el navegador, puede seleccionar la opción **Abrir Archivo** desde el menú del navegador o hacer doble clic en el archivo desde el explorador de archivos de Windows (o Finder en ordenadores Apple), y el sistema se encarga de abrir el navegador y cargar el documento.

Aunque la mayoría de los ejemplos de este libro se pueden probar sin subirlos a un servidor, abrir un sitio web completo en un ordenador personal no es siempre posible. Como veremos más adelante, algunos códigos JavaScript solo trabajan cuando se descargan desde un servidor, y tecnologías de servidor como PHP requieren ser alojadas en un servidor para funcionar. Para trabajar con estas clases de documentos existen dos alternativas: podemos obtener una cuenta de alojamiento web de inmediato y usarla para hacer pruebas, o instalar un servidor en nuestro propio ordenador. Esta última opción no hará que se pueda acceder a nuestro sitio web desde Internet, pero nos permite probarlo y experimentar con el código antes de subir la versión final a un servidor real.

Existen varios paquetes que instalan todos los programas necesarios para convertir nuestro ordenador en un servidor. Estos paquetes incluyen un servidor Apache (para despachar archivos web a través del protocolo HTTP), un servidor PHP (para procesar código PHP), y un servidor MySQL (para procesar bases de datos de tipo MySQL que podemos usar para almacenar datos en el servidor). El que recomendamos se llama *MAMP*. Es un paquete gratuito, disponible para ordenadores personales y ordenadores Apple, que podemos descargar desde **www.mamp.info** (la empresa también ofrece una versión comercial avanzada llamada *MAMP PRO*).

MAMP es fácil de instalar y usar. Una vez descargado e instalado, solo necesitamos abrirlo para comenzar a utilizar el servidor.

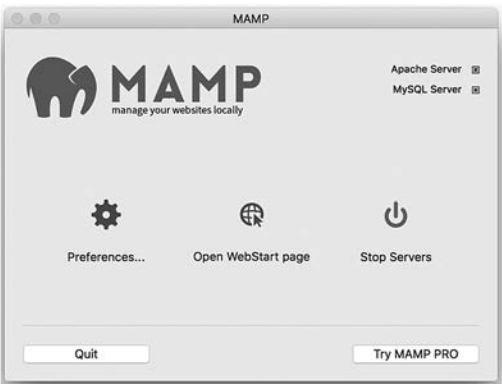

Figura 1-15: *Pantalla principal de MAMP*

MAMP crea un directorio dentro de su propio directorio llamado *htdocs* donde se supone que debemos almacenar los archivos de nuestro sitio web, pero si lo deseamos, podemos asignar un directorio diferente desde la opción **Preferences**. Esta opción abre una nueva ventana con varias pestañas para su configuración. La pestaña **Web Server** muestra el directorio actual que usa el servidor Apache y ofrece un botón para seleccionar uno diferente, tal como ilustra la Figura 1-16, número 1.

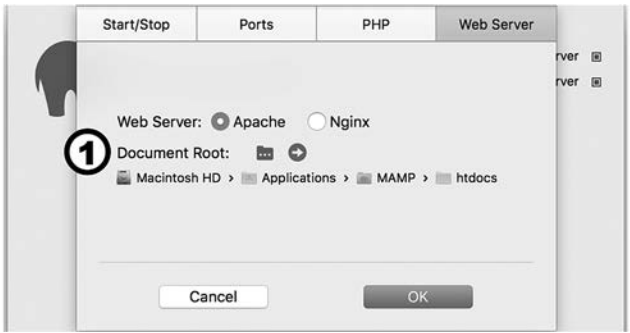

Figura 1-16: *Directorio del servidor Apache*

Después de seleccionar el directorio en el que se encuentran los archivos de nuestro sitio web, podemos acceder a ellos desde el servidor. Apache crea un dominio especial llamado *localhost* para referenciar al servidor y, por lo tanto, se puede acceder a nuestro sitio web desde la URL http://localhost/. Si queremos acceder a un archivo específico, solo tenemos que agregar el nombre del archivo al final de la URL, tal como hacemos con cualquier otro dominio (por ejemplo, http://localhost/contacto.html).

Hágalo usted mismo: visite **www.mamp.info** y descargue la versión gratuita de MAMP para su sistema (Windows o macOS). Instale el paquete y abra la aplicación. Seleccione la opción **Preferences** (Figura 1-15) y reemplace el directorio htdocs por el directorio que haya creado para su sitio web en el ejemplo anterior (Figura 1-8). Abra el navegador e inserte la URL http://localhost/. Si ha creado el archivo index.html como sugerimos anteriormente, debería ver el texto HOLA MUNDO! en la pantalla (Figura 1-14).

IMPORTANTE: el sistema operativo de Apple incluye su propia versión del servidor Apache, lo que obliga a MAMP a conectar Apache en un puerto diferente para evitar conflictos. Por esta razón, en un ordenador Mac tiene que especificar el puerto 8888 cuando intenta acceder al localhost (http://localhost:8888). Si lo desea, puede cambiar el puerto desde la configuración de MAMP. Haga clic en **Preferences**, seleccione la pestaña **Ports**, y presione el botón **Set Web & MySQL ports**.

Lo básico: la mayoría de los ejemplos de este libro se pueden ejecutar en un ordenador personal sin tener que instalar ningún servidor (le informaremos cuando esto no sea posible), pero si lo desea, puede instalar MAMP para asegurarse de que todo funcione correctamente como lo haría en un servidor real.

Capítulo 2
HTML

2.1 Estructura

A pesar de las innovaciones introducidas por CSS y JavaScript en estos últimos años, la estructura creada por el código HTML sigue siendo la parte fundamental del documento. Esta estructura define el espacio dentro del documento donde el contenido estático y dinámico es posicionado y es la plataforma básica para toda aplicación. Para crear un sitio o una aplicación web, lo primero que debemos hacer es programar el código HTML que define la estructura de cada una de las páginas que lo componen.

IMPORTANTE: los documentos HTML son archivos de texto que se pueden crear con cualquier editor de texto o los editores profesionales indicados en el Capítulo 1. La mayoría de estos editores ofrecen herramientas para ayudarle a escribir sus documentos, pero no controlan la validez del código. Si omite una etiqueta o se olvida de escribir una parte del código, el editor no le advierte sobre el error cometido. Para controlar el código de sus documentos, puede usar herramientas de validación en línea. La más popular para documentos HTML se encuentra disponible en http://validator.w3.org.

Tipo de documento

Debido a que los navegadores son capaces de procesar diferentes tipos de archivos, lo primero que debemos hacer en la construcción de un documento HTML es indicar su tipo. Para asegurarnos de que el contenido de nuestros documentos sea interpretado correctamente como código HTML, debemos agregar la declaración **<!DOCTYPE>** al comienzo del archivo. Esta declaración, similar en formato a las etiquetas HTML, se requiere al comienzo de cada documento para ayudar al navegador a decidir cómo debe generar la página web. Para documentos programados con HTML5, la declaración debe incluir el atributo **html**, según la definimos en el siguiente ejemplo.

```
<!DOCTYPE html>
```

Listado 2-1: *Incluyendo la declaración* <!DOCTYPE>

Hágalo usted mismo: abra Atom o su editor favorito y cree un nuevo archivo llamado index.html para probar los códigos de este capítulo (también puede usar el archivo creado en el capítulo anterior).

IMPORTANTE: la línea con la declaración **<!DOCTYPE>** debe ser la primera línea de su documento, sin ningún espacio o código previo. Esto activa el modo estándar y obliga a los navegadores a interpretar HTML5 cuando es posible o ignorarlo en caso contrario.

Lo básico: algunos de los elementos y atributos introducidos en HTML5 no están disponibles en viejos navegadores como Internet Explorer. Para saber qué navegadores implementan estos elementos u otras funciones incorporadas por HTML5, visite www.caniuse.com. Este sitio web ofrece una lista de todos los elementos, atributos, propiedades CSS, y códigos JavaScript disponibles en HTML5, junto con los navegadores que los admiten.

Elementos estructurales

Como mencionamos en el Capítulo 1, los elementos HTML conforman una estructura de tipo árbol con el elemento `<html>` como su raíz. Esta estructura presenta múltiples niveles de organización, con algunos elementos a cargo de definir secciones generales del documento y otros encargados de representar secciones menores o contenido. Los siguientes son los elementos disponibles para definir la columna vertebral de la estructura y facilitar la información que el navegador necesita para mostrar la página en la pantalla.

<html>—Este elemento delimita el código HTML. Puede incluir el atributo `lang` para definir el idioma del contenido del documento.

<head>—Este elemento se usa para definir la información necesaria para configurar la página web, como el título, el tipo de codificación de caracteres y los archivos externos requeridos por el documento.

<body>—Este elemento delimita el contenido del documento (la parte visible de la página).

Después de declarar el tipo de documento, tenemos que construir la estructura de tipo árbol con los elementos HTML, comenzando por el elemento `<html>`. Este elemento puede incluir el atributo `lang` para declarar el idioma en el que vamos a escribir el contenido de la página, tal como muestra el siguiente ejemplo.

```
<!DOCTYPE html>
<html lang="es">

</html>
```

Listado 2-2: *Incluyendo el elemento* `<html>`

Lo básico: existen varios valores disponibles para el atributo `lang`, incluidos `en` para inglés, `es` para español, `fr` para francés, entre otros. Para obtener una lista completa, visite https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes.

El código HTML insertado entre las etiquetas `<html>` se tiene que dividir en dos secciones principales: la cabecera y el cuerpo. Por supuesto, la cabecera va primero y, al igual que el resto de los elementos estructurales, está compuesta por etiquetas de apertura y cierre.

```
<!DOCTYPE html>
<html lang="es">
<head>

</head>
</html>
```

Listado 2-3: *Incluyendo el elemento* `<head>`

Entre las etiquetas **<head>** debemos definir el título de la página web, declarar el tipo de codificación de caracteres, facilitar información general acerca del documento, e incorporar los archivos externos con estilos y códigos necesarios para generar la página. Excepto por el título e iconos, el resto de la información insertada en medio de estas etiquetas no es visible para el usuario.

La otra sección que forma parte de la organización principal de un documento HTML es el cuerpo, la parte visible del documento que se especifica con el elemento **<body>**.

```
<!DOCTYPE html>
<html lang="es">
<head>

</head>
<body>

</body>
</html>
```

Listado 2-4: *Incluyendo el elemento* `<body>`

Lo básico: como ya mencionamos, la estructura HTML puede describirse como un árbol, con el elemento **<html>** como su raíz, pero otra forma de definir la relación entre los elementos es describirlos como padres, hijos o hermanos, de acuerdo a sus posiciones en la estructura. Por ejemplo, en un documento HTML típico, el elemento **<body>** es hijo del elemento **<html>** y hermano del elemento **<head>**. Ambos, **<body>** y **<head>**, tienen al elemento **<html>** como su padre.

La estructura básica ya está lista. Ahora tenemos que construir la página, comenzando por la definición de la cabecera. La cabecera incluye toda la información y los recursos necesarios para generar la página. Los siguientes son los elementos disponibles para este propósito.

<title>—Este elemento define el título de la página.

<base>—Este elemento define la URL usada por el navegador para establecer la ubicación real de las URL relativas. El elemento debe incluir el atributo **href** para declarar la URL base. Cuando se declara este elemento, en lugar de la URL actual, el navegador usa la URL asignada al atributo **href** para completar las URL relativas.

<meta>—Este elemento representa metadatos asociados con el documento, como la descripción del documento, palabras claves, el tipo de codificación de caracteres, etc. El elemento puede incluir los atributos **name** para describir el tipo de metadata, **content**

para especificar el valor, y **`charset`** para declarar el tipo de codificación de caracteres a utilizar para procesar el contenido.

<link>—Este elemento especifica la relación entre el documento y un recurso externo (generalmente usado para cargar archivos CSS). El elemento puede incluir los atributos **`href`** para declarar la ubicación del recurso, **`rel`** para definir el tipo de relación, **`media`** para especificar el medio al que el recurso está asociado (pantalla, impresora, etc.), y **`type`** y **`sizes`** para declarar el tipo de recurso y su tamaño (usado a menudo para cargar iconos).

<style>—Este elemento se usa para declarar estilos CSS dentro del documento (estudiado en el Capítulo 3).

<script>—Este elemento se usa para cargar o declarar código JavaScript (estudiado en el Capítulo 6).

Lo primero que tenemos que hacer cuando declaramos la cabecera del documento es especificar el título de la página con el elemento **`<title>`**. Este texto es el que muestran los navegadores en la parte superior de la ventana, y es lo que los usuarios ven cuando buscan información en nuestro sitio web por medio de motores de búsqueda como Google o Yahoo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
</head>
<body>

</body>
</html>
```

Listado 2-5: *Incluyendo el elemento* `<title>`

Hágalo usted mismo: reemplace el código en su archivo index.html por el código del Listado 2-5 y abra el documento en su navegador (para abrirlo, puede hacer doble clic en el archivo o seleccionar la opción **Abrir Archivo** desde el menú **Archivos** en su navegador). Debería ver el texto especificado entre las etiquetas **`<title>`** en la parte superior de la ventana.

Lo básico: el elemento **`<title>`** en el ejemplo del Listado 2-5 se ha desplazado hacia la derecha. El espacio en blanco en el lado izquierdo se usa para ayudar al desarrollador a visualizar la posición del elemento dentro de la jerarquía del documento Este espacio se genera automáticamente por editores como Atom, pero puede hacerlo usted mismo cuando lo necesite pulsando la tecla Tab (Tabulador) en su teclado (los navegadores ignoran los espacios en blanco y los saltos de líneas que se encuentran fuera de los elementos).

Además del título, también tenemos que declarar los metadatos del documento. Los metadatos incluyen información acerca de la página que los navegadores, y también los motores de búsqueda, utilizan para generar y clasificar la página web. Los valores se declaran con el elemento **`<meta>`**. Este elemento incluye varios atributos, pero cuáles usemos

dependerá del tipo de información que queremos declarar. Por ejemplo, el valor más importante es el que define la tabla de caracteres a utilizar para presentar el texto en pantalla, el cual se declara con el atributo **charset**.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
</head>
<body>

</body>
</html>
```

Listado 2-6: *Incluyendo el elemento* <meta>

Lo básico: el ejemplo del Listado 2-6 define el grupo de caracteres como **utf-8**, que es el que se recomienda debido a que incluye todos los caracteres utilizados en la mayoría de los idiomas, pero existen otros disponibles. Para más información, visite nuestro sitio web y siga los enlaces de este capítulo.

Se pueden incluir múltiples elementos **<meta>** para declarar información adicional. Por ejemplo, dos datos que los navegadores pueden considerar a la hora de procesar nuestros documentos son la descripción de la página y las palabras claves que identifican su contenido. Estos elementos **<meta>** requieren el atributo **name** con los valores "description" y "keywords", y el atributo **content** con el texto que queremos asignar como descripción y palabras clave (las palabras clave se deben separar por comas).

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
</head>
<body>

</body>
</html>
```

Listado 2-7: *Agregando información adicional con el elemento* <meta>

Otro elemento importante de la cabecera del documento es **<link>**. Este elemento se usa para incorporar al documento estilos, códigos, imágenes o iconos desde archivos externos. Por ejemplo, algunos navegadores muestran un icono en la parte superior de la ventana junto con el título de la página. Para cargar este icono, tenemos que incluir un elemento **<link>** con el atributo **rel** definido como **icon**, el atributo **href** con la ubicación del archivo que contiene

el icono, el atributo **type** para especificar el formato con el que se ha creado el icono, y el atributo **sizes** con el ancho y la altura del icono separados por la letra x.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="icon" href="imagenes/favicon.png" type="image/png"
sizes="16x16">
</head>
<body>

</body>
</html>
```

Listado 2-8: *Incluyendo el icono del documento*

El navegador tiene poco espacio para mostrar el icono, por lo tanto el tamaño típico de esta imagen es de unos 16 píxeles por 16 píxeles. La Figura 2-1 muestra cómo se ve la ventana cuando abrimos un documento que contiene un icono (en este caso, se muestra una imagen con la letra M en el lado izquierdo del título).

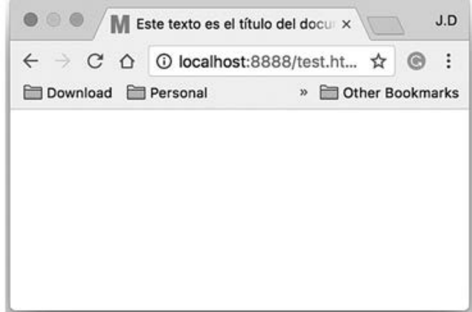

Figura 2-1: *El icono del documento en el navegador*

Hágalo usted mismo: actualice el código en su archivo index.html con el ejemplo del Listado 2-8. El icono se carga desde el archivo favicon.png que debe copiar dentro del directorio de su proyecto. Puede descargar este archivo desde nuestro sitio web o usar el suyo.

Lo básico: el valor asignado al atributo **type** del elemento **<link>** debe ser especificado como un tipo MIME. Todo archivo tiene un tipo MIME asociado para indicar al sistema el formato de su contenido. Por ejemplo, el tipo MIME de un archivo HTML es **text/html**. Existe un tipo MIME para cada tipo de archivo disponible, que incluye **image/jpeg** e **image/png** para imágenes JPEG y PNG. Para obtener una lista completa, visite nuestro sitio web y siga los enlaces de este capítulo.

El elemento **<link>** se usa comúnmente para cargar archivos CSS con los estilos necesarios para generar la página web. Por ejemplo, el siguiente documento carga el archivo misestilos.css. Después de cargar el archivo, todos los estilos declarados en su interior se aplican a los elementos del documento. En este caso, solo necesitamos incluir el atributo **rel** para declarar el tipo de recurso (para hojas de estilo CSS debemos asignar el valor "stylesheet") y el atributo **href** con la URL que determina la ubicación del archivo (estudiaremos cómo crear esta clase de archivos y definir estilos CSS en el Capítulo 3).

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>

</body>
</html>
```

Listado 2-9: *Cargando un archivo CSS con el elemento* <link>

Con la cabecera lista, es hora de construir el cuerpo. Esta estructura (el código entre las etiquetas **<body>**) es la encargada de generar la parte visible de nuestro documento (la página web).

HTML siempre ha ofrecido diferentes maneras de construir y organizar la información en el cuerpo del documento. Uno de los primeros elementos utilizados con este propósito fue **<table>** (tabla). Este elemento permitía a los desarrolladores organizar datos, textos, imágenes, así como herramientas en filas y columnas de celdas. Con la introducción de CSS, la estructura generada por estas tablas ya no resultaba práctica, por lo que los desarrolladores comenzaron a implementar un elemento más flexible llamado **<div>** (división). Pero **<div>**, así como **<table>**, no facilita demasiada información acerca de las partes del cuerpo que representa. Cualquier cosa, desde imágenes hasta menús, texto, enlaces, códigos o formularios, se puede insertar entre las etiquetas de apertura y cierre de un elemento **<div>**. En otras palabras, el nombre **div** solo especifica una división en el cuerpo, como una celda en una tabla, pero no ofrece ninguna pista acerca del tipo de división que está creando, cuál es su propósito o qué contiene. Esta es la razón por la que HTML5 introdujo nuevos elementos con nombres más descriptivos que permiten a los desarrolladores identificar cada parte del documento. Estos elementos no solo ayudan a los desarrolladores a crear el documento, sino que además informan al navegador sobre el propósito de cada sección. La siguiente lista incluye todos los elementos disponibles para definir la estructura del cuerpo.

<div>—Este elemento define una división genérica. Se usa cuando no se puede aplicar ningún otro elemento.

<main>—Este elemento define una división que contiene el contenido principal del documento (el contenido que representa el tema central de la página).

<nav>—Este elemento define una división que contiene ayuda para la navegación, como el menú principal de la página o bloques de enlaces necesarios para navegar en el sitio web.

<section>—Este elemento define una sección genérica. Se usa frecuentemente para separar contenido temático, o para generar columnas o bloques que ayudan a organizar el contenido principal.

<aside>—Este elemento define una división que contiene información relacionada con el contenido principal pero que no es parte del mismo, como referencias a artículos o enlaces que apuntan a publicaciones anteriores.

<article>—Este elemento representa un artículo independiente, como un mensaje de foro, el artículo de una revista, una entrada de un blog, un comentario, etc.

<header>—Este elemento define la cabecera del cuerpo o de secciones dentro del cuerpo.

<footer>—Este elemento define el pie del cuerpo o de secciones dentro del cuerpo.

Estos elementos han sido definidos con el propósito de representar secciones específicas de una página web. Aunque son flexibles y se pueden implementar en diferentes partes del diseño, todos siguen un patrón que se encuentra comúnmente en la mayoría de los sitios web. La Figura 2-2, a continuación, ilustra este tipo de diseño.

Figura 2-2: *Representación visual de un diseño web tradicional*

A pesar de que cada desarrollador crea sus propios diseños, en general podremos describir todo sitio web considerando estas secciones. En la barra superior, descrita como **cabecera** en la Figura 2-2, ubicamos el logo, el nombre del sitio, los subtítulos y una descripción breve de nuestro sitio o página web. En la **barra de navegación** situada debajo es donde la mayoría de los desarrolladores ofrecen un menú o una lista de enlaces para navegar en el sitio. El contenido relevante de la página se ubica en el medio del diseño, donde generalmente encontramos artículos o noticias, y también enlaces a documentos relacionados o recursos. En el ejemplo de la Figura 2-2, esta sección se ha dividido en dos columnas, **información principal** y **barra lateral**, pero los diseñadores la adaptan a sus necesidades insertando columnas adicionales o dividiendo las columnas en bloques más pequeños. En la parte inferior de un diseño tradicional, nos encontramos con otra barra llamada **barra institucional**. La llamamos de este modo porque en este área es

donde mostramos información general acerca del sitio web, el autor, la compañía, los enlaces relacionados con reglas de uso, términos y condiciones, el mapa del sitio, etc.

Como mencionamos anteriormente, los elementos de HTML5 se han diseñado siguiendo este patrón. En la Figura 2-3 aplicamos los elementos introducidos anteriormente para definir el diseño de la Figura 2-2.

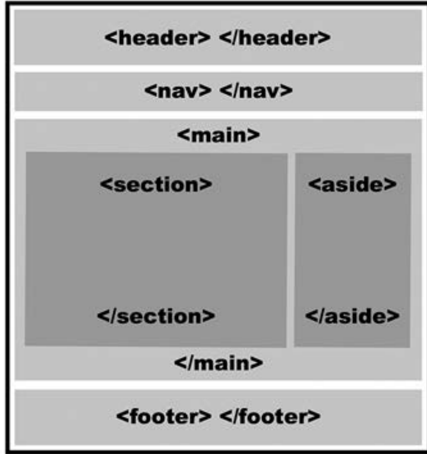

Figura 2-3: *Representación de la estructura de un documento usando elementos de HTML5*

Los elementos se declaran en el documento en el mismo orden en el que se presentarán en pantalla, desde la parte superior a la inferior y de izquierda a derecha (este orden se puede modificar por medio de estilos CSS, como veremos en el Capítulo 4). El primer elemento de un diseño tradicional es **<header>**. No debemos confundir este elemento con el elemento **<head>** utilizado anteriormente para crear la cabecera del documento. Al igual que **<head>**, el elemento **<header>** se ha definido para facilitar información introductoria, como títulos o subtítulos, pero no para el documento, sino para el cuerpo o secciones dentro del cuerpo del documento. En el siguiente ejemplo, este elemento se usa para definir el título de la página web.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <header>
    Este es el título
  </header>
</body>
</html>
```

Listado 2-10: *Incluyendo el elemento* <header>

La inserción del elemento **<header>** representa el comienzo del cuerpo y de la parte visible del documento. De ahora en adelante, podremos ver el resultado de la ejecución del código en la ventana del navegador.

Hágalo usted mismo: reemplace el código en su archivo index.html por el código del Listado 2-10 y abra el documento en su navegador. Debería ver el título de la página en la pantalla.

La siguiente sección de nuestro ejemplo es la **barra de navegación**. Esta barra define una sección con ayuda para la navegación y se representa con el elemento **<nav>**.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <header>
    Este es el título
  </header>
  <nav>
    Principal | Fotos | Videos | Contacto
  </nav>
</body>
</html>
```

Listado 2-11: *Incluyendo el elemento* <nav>

La estructura y el orden que decidimos implementar depende de lo que nuestro sitio web o aplicación requieran. Los elementos HTML son bastante flexibles y solo nos dan ciertos parámetros con los que trabajar, pero el modo en que los usemos depende de nosotros. Un ejemplo de esta versatilidad es que el elemento **<nav>** se podría insertar dentro de etiquetas **<header>** o en otra sección del cuerpo. Sin embargo, siempre debemos considerar que estos elementos se han creado para ofrecer información adicional al navegador, y ayudar a cada nuevo programa y dispositivo a identificar las partes relevantes del documento. Si queremos mantener nuestro código HTML portable y legible, es mejor seguir los estándares establecidos por estos elementos. El elemento **<nav>** se ha creado con la intención de contener ayuda para la navegación, como el menú principal o bloques de enlaces importantes, y deberíamos usarlo con este propósito.

Otro ejemplo de especificidad es el que ofrecen los elementos **<main>**, **<section>**, y **<aside>**, que se han diseñado para organizar el contenido principal del documento. En nuestro diseño, estos elementos representan las secciones que llamamos **Información principal** y **Barra lateral**. Debido a que la sección **Información principal** abarca más, su contenido generalmente se representa por elementos **<section>** (uno o varios, dependiendo del diseño), y debido al tipo de información que contiene, el elemento **<aside>** se ubica en los laterales de la página. La mayoría del tiempo, estos dos elementos son suficientes para

representar el contenido principal, pero como se pueden usar en otras áreas del documento, se implementa el elemento **<main>** para agruparlos, como lo muestra el siguiente ejemplo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <header>
    Este es el título
  </header>
  <nav>
    Principal | Fotos | Videos | Contacto
  </nav>
  <main>
    <section>
      Artículos
    </section>
    <aside>
      Cita del artículo uno
      Cita del artículo dos
    </aside>
  </main>
</body>
</html>
```

Listado 2-12: *Organizando el contenido principal*

El elemento **<aside>** describe la información que contiene, no un lugar en la estructura, por lo que se podría ubicar en cualquier parte del diseño, y se puede usar mientras su contenido no se considere el contenido principal del documento.

IMPORTANTE: los elementos que representan cada sección del documento se listan en el código uno encima del otro, pero en la página web algunas de estas secciones se muestran una al lado de la otra (por ejemplo, las columnas creadas por las secciones **Información principal** y **Barra lateral** se muestran en una misma línea en la página web). Si abre el documento del Listado 2-12 en su navegador, verá que los textos se muestran uno por línea. Esto se debe a que HTML5 delega la tarea de presentar el documento a CSS. Para mostrar las secciones en el lugar correcto, debemos asignar estilos CSS a cada elemento del documento. Estudiaremos CSS en los Capítulos 3 y 4.

El diseño considerado anteriormente (Figura 2-2) es el más común de todos y representa la estructura básica de la mayoría de los sitios web que encontramos hoy en día, pero es también una muestra de cómo se muestra el contenido importante de una página web en pantalla.

Como los artículos de un diario, las páginas web generalmente presentan la información dividida en secciones que comparten características similares. El elemento **<article>** nos permite identificar cada una de estas partes. En el siguiente ejemplo, implementamos este elemento para representar las publicaciones que queremos mostrar en la sección principal de nuestra página web.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <header>
    Este es el título
  </header>
  <nav>
    Principal | Fotos | Videos | Contacto
  </nav>
  <main>
    <section>
      <article>
        Este es el texto de mi primer artículo
      </article>
      <article>
        Este es el texto de mi segundo artículo
      </article>
    </section>
    <aside>
      Cita del artículo uno
      Cita del artículo dos
    </aside>
  </main>
</body>
</html>
```

Listado 2-13: *Incluyendo el elemento* <article>

En este punto, ya contamos con la cabecera y el cuerpo del documento, secciones con ayuda para la navegación y el contenido, e información adicional a un lado de la página. Lo único que nos queda por hacer es cerrar el diseño y finalizar el cuerpo del documento. Con este fin, HTML ofrece el elemento **<footer>**.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
```

```
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <header>
    Este es el título
  </header>
  <nav>
    Principal | Fotos | Videos | Contacto
  </nav>
  <main>
    <section>
      <article>
        Este es el texto de mi primer artículo
      </article>
      <article>
        Este es el texto de mi segundo artículo
      </article>
    </section>
    <aside>
      Cita del artículo uno
      Cita del artículo dos
    </aside>
  </main>
  <footer>
    &copy; Derechos Reservados 2016
  </footer>
</body>
</html>
```

Listado 2-14: *Incluyendo el elemento* `<footer>`

En un diseño web tradicional (Figura 2-2), la sección **Barra institucional** se define con el elemento **`<footer>`**. Esto se debe a que la sección representa el final (o pie) de nuestro documento y se usa comúnmente para compartir información general acerca del autor del sitio o la compañía detrás del proyecto, como derechos de autor, términos y condiciones, etc.

El elemento **`<footer>`** se usa para representar el final del documento y tiene el objetivo principal ya mencionado, sin embargo, este elemento y el elemento **`<header>`** también se pueden utilizar dentro del cuerpo para representar el comienzo y final de una sección.

Lo básico: el ejemplo del Listado 2-14 incluye la cadena de caracteres **`©`** al pie del documento. Sin embargo, cuando se carga el documento, el navegador reemplaza estos caracteres por el carácter de derechos de autor (©). Estas cadenas de caracteres se denominan *entidades* (*character entities*) y representan caracteres especiales que no se encuentran en el teclado o tienen un significado especial en HTML, como el carácter de derechos de autor (©), el de marca registrada (®) o los paréntesis angulares usados por HTML para definir los elementos (< y >). Cuando necesite incluir en sus textos uno de estos caracteres, debe escribir la entidad en su lugar. Por ejemplo, si quiere incluir los caracteres < y > dentro de un texto, debe representarlos con las cadenas de caracteres **`<`** y **`>`**. Otras entidades de uso común son **`&`** (&), **`"`** ("), **`'`** ('), **`£`** (£), y

€ (€). Para obtener una lista completa, visite nuestro sitio web y siga los enlaces de este capítulo.

Atributos globales

Aunque la mayoría de los elementos estructurales tienen un propósito implícito que se refleja en sus nombres, esto no significa que se deban usar solo una vez en el mismo documento. Por ejemplo, algunos elementos como **<section>** y **<aside>** se pueden utilizar muchas veces para representar diferentes partes de la estructura, y otros como **<div>** aún son implementados de forma repetida para separar contenido dentro de secciones. Por esta razón, HTML define atributos globales que podemos usar para asignar identificadores personalizados a cada elemento.

id—Este atributo nos permite asignar un identificador único a un elemento.

class—Este atributo asigna el mismo identificador a un grupo de elementos.

El atributo **id** identifica elementos independientes con un valor único, mientras que el valor del atributo **class** se puede duplicar para asociar elementos con características similares. Por ejemplo, si tenemos dos o más elementos **<section>** que necesitamos diferenciar entre sí, podemos asignar el atributo **id** a cada uno con valores que declaran sus propósitos.

```
<main>
  <section id="noticias">
    Artículos largos
  </section>
  <section id="noticiaslocales">
    Artículos cortos
  </section>
  <aside>
    Quote from article one
    Quote from article two
  </aside>
</main>
```

Listado 2-15: *Identificando elementos con el atributo* id

El ejemplo del Listado 2-15 incluye dos elementos **<section>** en la sección principal del documento para separar artículos de acuerdo a su extensión. Debido a que el contenido de estos elementos es diferente, requieren distintos estilos y, por lo tanto, tenemos que identificarlos con diferentes valores. El primer elemento **<section>** se ha identificado con el valor "noticias" y el segundo elemento con el valor "noticiaslocales".

Por otro lado, si lo que necesitamos es identificar un grupo de elementos con características similares, podemos usar el atributo **class**. El siguiente ejemplo divide el contenido de una sección con elementos **<div>**. Debido a que todos tienen un contenido similar, compartirán los mismos estilos y, por lo tanto, deberíamos identificarlos con el mismo valor (todo son de la misma clase).

```
<main>
  <section>
    <div class="libros">Libro: IT, Stephen King</div>
    <div class="libros">Libro: Carrie, Stephen King</div>
    <div class="libros">Libro: El resplandor, Stephen King</div>
    <div class="libros">Libro: Misery, Stephen King</div>
  </section>
  <aside>
    Cita del artículo uno
    Cita del artículo dos
  </aside>
</main>
```

Listado 2-16: *Identificando elementos con el atributo* `class`

En el código del Listado 2-16, tenemos un único elemento **`<section>`** con el que representamos el contenido principal del documento, pero hemos creado varias divisiones con elementos **`<div>`** para organizar el contenido. Debido a que estos elementos se han identificado con el atributo **`class`** y el valor "libros", cada vez que accedemos o modificamos elementos referenciando la clase **`libros`**, todos estos elementos se ven afectados.

IMPORTANTE: los valores de los atributos **`id`** y **`class`** son usados por algunos elementos HTML para identificar otros elementos y también por reglas CSS y código JavaScript para acceder y modificar elementos específicos en el documento. En próximos capítulos veremos algunos ejemplos prácticos.

Lo básico: los valores asignados a estos atributos son arbitrarios. Puede asignarles cualquier valor que desee, siempre y cuando no incluya ningún espacio en blanco (el atributo **`class`** utiliza espacios para asignar múltiples clases a un mismo elemento). Así mismo, para mantener su sitio web compatible con todos los navegadores, debería usar solo letras y números, siempre comenzar con una letra y evitar caracteres especiales.

2.2 Contenido

Hemos concluido la estructura básica de nuestro sitio web, pero todavía tenemos que trabajar en el contenido. Los elementos HTML estudiados hasta el momento nos ayudan a identificar cada sección del diseño y asignarles un propósito, pero lo que realmente importa en una página web es lo que hay dentro de esas secciones. Debido a que esta información está compuesta por diferentes elementos visuales, como títulos, textos, imágenes y vídeos, entre otros, HTML define varios elementos para representarla.

Texto

El medio más importante que puede incluir un documento es texto. HTML define varios elementos para determinar el propósito de cada palabra, frase, o párrafo en el documento. El siguiente elemento se usa para representar títulos.

<h1>—Este elemento representa un título. El título se declara entre las etiquetas de apertura y cierre. HTML también incluye elementos adicionales para representar subtítulos, hasta seis niveles (**<h2>**, **<h3>**, **<h4>**, **<h5>**, y **<h6>**).

Cada vez que queremos insertar un título o un subtítulo en el documento, tenemos que incluirlo dentro de etiquetas **<h>**. Por ejemplo, el documento que hemos creado en la sección anterior incluye el título de la página. Como este es el título principal, debería representarse con el elemento **<h1>**, tal como ilustra el siguiente ejemplo.

```
<header>
  <h1>Este es el título</h1>
</header>
```

Listado 2-17: *Incluyendo el elemento* <h1>

Los navegadores otorgan estilos por defecto a los elementos **<h>** que incluyen márgenes y diferentes tamaños de letras, dependiendo de la jerarquía (**<h1>** es el de más alta jerarquía y **<h6>** el de menor jerarquía). La Figura 2-4, debajo, muestra cómo se ve el texto dentro de un elemento **<h1>** con los estilos por defecto.

Este es el título

Principal | Fotos | Videos | Contacto
Este es el texto de mi primer artículo
Este es el texto de mi segundo artículo
Cita del artículo uno Cita del artículo dos
© Derechos Reservados 2016

Figura 2-4: *Título representado por un elemento* <h1> *con estilos por defecto*

Hágalo usted mismo: reemplace el elemento **<header>** en su archivo index.html por el código del Listado 2-17. Abra el documento en su navegador. Debería ver algo similar a lo que se muestra en la Figura 2-4.

Los siguientes son los elementos que ofrece HTML para representar el cuerpo del texto.

<p>—Este elemento representa un párrafo. Por defecto, los navegadores le asignan un margen en la parte superior para separar un párrafo de otro.

<pre>—Este elemento representa un texto con formato predefinido, como código de programación o un poema que requiere que los espacios asignados a cada carácter y los saltos de línea se muestren como se han declarado originalmente.

<span>—Este elemento puede contener un párrafo, una frase o una palabra. No aplica ningún estilo al texto pero se usa para asignar estilos personalizados, como veremos en próximos capítulos.

El elemento **<p>** se utiliza ampliamente para representar el cuerpo del texto. Por defecto, los navegadores les asignan estilos que incluyen márgenes y un salto de línea para diferenciar

un párrafo de otro. Debido a estas características, también podemos utilizar los elementos **<p>** para dar formato a líneas de texto, como las citas de nuestro ejemplo.

```
<aside>
  <p>Cita del artículo uno</p>
  <p>Cita del artículo dos</p>
</aside>
```

Listado 2-18: *Definiendo líneas de texto con el elemento* <p>

El Listado 2-18 presenta las citas dentro del elemento **<aside>** de ejemplos anteriores con elementos **<p>**. Ahora, el navegador muestra cada cita en una línea distinta.

Este es el título

Principal | Fotos | Videos | Contacto
Este es el texto de mi primer artículo
Este es el texto de mi segundo artículo

Cita del artículo uno

Cita del artículo dos

© Derechos Reservados 2016

Figura 2-5: *Líneas de texto definidas con elementos* <p>

Cuando un párrafo incluye múltiples espacios, el elemento **<p>** automáticamente reduce ese espacio a solo un carácter e ignora el resto. El elemento también hace algo similar con los saltos de línea. Todo salto de línea introducido en el documento no se considera cuando el texto se muestra en la pantalla. Si queremos que estos espacios y saltos de línea se muestren al usuario, en lugar de usar el elemento **<p>** tenemos que usar el elemento **<pre>**.

```
<article>
  <pre>
    La muerte es una quimera: porque mientras yo existo, no existe la
muerte;
    y cuando existe la muerte, ya no existo yo.
    Epicuro de Samos
  </pre>
</article>
```

Listado 2-19: *Mostrando texto en su formato original*

El ejemplo del Listado 2-19 define un elemento **<article>** que contiene una cita de Epicuro de Samos. Como usamos el elemento **<pre>**, los saltos de línea son considerados por el navegador y las frases se muestran una por línea, tal como se han definido en el código.

```
La muerte es una quimera: porque mientras yo existo, no existe la muerte;
y cuando existe la muerte, ya no existo yo.
Epicuro de Samos
```

Figura 2-6: *Texto introducido con un elemento* `<pre>`

Hágalo usted mismo: reemplace el elemento **`<article>`** en su archivo index.html por el código del Listado 2-19. Abra el documento en su navegador. Debería ver algo similar a lo que se muestra en la Figura 2-6.

El elemento **`<pre>`** se configura por defecto con márgenes y un tipo de letra que respeta el formato asignado al texto original, lo que lo hace apropiado para presentar código de programación y cualquier clase de texto con formato predefinido. En casos como el del ejemplo anterior, donde lo único que necesitamos es incluir saltos de línea dentro del párrafo, podemos usar otros elementos que se han diseñado específicamente con este propósito.

**
**—Este elemento se usa para insertar saltos de línea.

<wbr>—Este elemento sugiere la posibilidad de un salto de línea para ayudar al navegador a decidir dónde cortar el texto cuando no hay suficiente espacio para mostrarlo entero.

Estos elementos se insertan dentro del texto para generar saltos de línea. Por ejemplo, podemos escribir el párrafo anterior en una sola línea e insertar elementos **`<br>`** al final de cada frase para presentarlas en líneas aparte.

```
<article>
  <p>La muerte es una quimera: porque mientras yo existo, no existe la
muerte;<br>y cuando existe la muerte, ya no existo yo.<br>Epicuro de
Samos</p>
</article>
```

Listado 2-20: *Agregando saltos de línea a un párrafo con el elemento* `<br>`

A diferencia de los elementos **`<p>`** y **`<pre>`**, los elementos **`<br>`** y **`<wbr>`** no asignan ningún margen o tipo de letra al texto, por lo que las líneas se muestran como si pertenecieran al mismo párrafo y con el tipo de letra definida por defecto.

La muerte es una quimera: porque mientras yo existo, no existe la muerte;
y cuando existe la muerte, ya no existo yo.
Epicuro de Samos

Figura 2-7: *Saltos de línea generadas por elementos* `<br>`

Debido a que no todas las palabras en un texto tienen el mismo énfasis, HTML incluye los siguientes elementos para declarar un significado especial a palabras individuales o frases completas.

<em>—Este elemento se usa para indicar énfasis. El texto se muestra por defecto con letra cursiva.

<strong>—Este elemento es utiliza para indicar importancia. El texto se muestra por defecto en negrita.

<i>—Este elemento representa una voz alternativa o un estado de humor, como un pensamiento, un término técnico, etc. El texto se muestra por defecto con letra cursiva.

<u>—Este elemento representa texto no articulado. Por defecto se muestra subrayado.

<b>—Este elemento se usa para indicar importancia. Debería ser implementado solo cuando ningún otro elemento es apropiado para la situación. El texto se muestra por defecto en negrita.

Estos elementos se pueden utilizar para resaltar títulos o etiquetas, o para destacar palabras o frases en un párrafo, según muestra el siguiente ejemplo.

```
<article>
  <p>La muerte es una <em>quimera</em>: porque mientras yo existo, no existe la <i>muerte</i>;<br>y cuando existe la <i>muerte</i>, <strong>ya no existo yo</strong>.<br>Epicuro de Samos</p>
</article>
```

Listado 2-21: *Resaltando texto*

A menos que especifiquemos diferentes estilos con CSS, el texto dentro de estos elementos se muestra con los estilos por defecto, como ilustra la Figura 2-8.

La muerte es una *quimera*: porque mientras yo existo, no existe la *muerte*;
y cuando existe la *muerte*, **ya no existo yo**.
Epicuro de Samos

Figura 2-8: *Texto resaltado*

La especificidad de elementos estructurales también se manifiesta en algunos de los elementos utilizados para definir el contenido. Por ejemplo, HTML incluye los siguientes elementos para insertar textos que tienen un propósito claramente definido.

<mark>—Este elemento resalta texto que es relevante en las circunstancias actuales (por ejemplo, términos que busca el usuario).

<small>—Este elemento representa letra pequeña, como declaraciones legales, descargos, etc.

<cite>—Este elemento representa el autor o título de una obra, como un libro, una película, etc.

<address>—Este elemento representa información de contacto. Se implementa con frecuencia dentro de los pies de página para definir la dirección de la empresa o el sitio web.

<time>—Este elemento representa una fecha en formato legible para el usuario. Incluye el atributo **datetime** para especificar un valor en formato de ordenador y el atributo **pubdate**, el cual indica que el valor asignado al atributo **datetime** representa la fecha de publicación.

<code>—Este elemento representa código de programación. Se usa en conjunto con el elemento **<pre>** para presentar código de programación en el formato original.

<data>—Este elemento representa datos genéricos. Puede incluir el atributo **value** para especificar el valor en formato de ordenador (por ejemplo, **<data value="32">Treinta y Dos</data>**).

Como estos elementos representan información específica, normalmente se utilizan para complementar el contenido de otros elementos. Por ejemplo, podemos usar el elemento **<time>** para declarar la fecha en la que un artículo se ha publicado y otros elementos como **<mark>** y **<cite>** para otorgarle significado a algunas partes del texto.

```
<article>
  <header>
    <h1>Título del artículo</h1>
    <time datetime="2016-10-12" pubdate>publicado 12-10-2016</time>
  </header>
  <p>La muerte es una quimera: porque mientras yo <mark>existo</mark>,
no existe la muerte;<br>y cuando existe la muerte, ya no existo
yo.<br><cite>Epicuro de Samos</cite></p>
</article>
```

Listado 2-22: *Complementando el elemento* `<article>`

El Listado 2-22 expande el elemento **<article>** utilizado en ejemplos anteriores. Este elemento ahora incluye un elemento **<header>** con el título del artículo y la fecha de publicación, y el texto se ha resaltado con los elementos **<mark>** y **<cite>**. El elemento **<mark>** resalta partes del texto que originalmente no se consideraban importantes, pero que al momento se han vuelto relevantes (quizás debido a que el usuario realizó una búsqueda con ese texto), y el elemento **<cite>**, en este caso, resalta el nombre del autor. Por defecto, los navegadores asignan estilos al texto dentro del elemento **<mark>** que incluyen un fondo amarillo y muestran el contenido del elemento **<cite>** en cursiva, tal como ilustra la siguiente figura.

Título del articulo

publicado 12-10-2016

La muerte es una quimera: porque mientras yo existo, no existe la muerte;
y cuando existe la muerte, ya no existo yo.
Epicuro de Samos

Figura 2-9: *Complementando el elemento* `<article>`

IMPORTANTE: el valor del atributo **datetime** del elemento **<time>** se debe declarar en formato de ordenador. Este formato requiere la sintaxis **2016-10-12T12:10:45**, donde la T se puede reemplazar por un espacio en blanco y la parte menos significativa se puede ignorar (por ejemplo, **2016-10-12**).

Lo básico: el atributo **pubdate** es un atributo booleano. Este tipo de atributos no requieren un valor; representan el valor **true** (verdadero) cuando están presentes o **false** (falso) en caso contrario.

El resto de los elementos mencionados arriba también se combinan con otros elementos para complementar sus contenidos. Por ejemplo, los elementos **<address>** y **<small>** se insertan normalmente dentro de un elemento **<footer>** para resaltar información acerca de la página o una sección.

```
<footer>
  <address>Toronto, Canada</address>
  <small>&copy; Derechos Reservados 2016</small>
</footer>
```

Listado 2-23: *Complementando el elemento* <footer>

Toronto, Canada
© Derechos Reservados 2016

Figura 2-10: *Complementando el elemento* <footer>

El elemento **<code>** también trabaja junto con otros elementos para presentar contenido, pero tiene una relación particular con el elemento **<pre>**. Estos elementos se implementan juntos para presentar código de programación. El elemento **<code>** indica que el contenido es código de programación y el elemento **<pre>** formatea ese contenido para mostrarlo en pantalla como se ha declarado originalmente en el documento (respetando los espacios y los saltos de línea).

```
<article>
  <pre>
    <code>
function cambiarColor() {
  document.body.style.backgroundColor = "#0000FF";
}
document.addEventListener("click", cambiarColor);
    </code>
  </pre>
</article>
```

Listado 2-24: *Presentando código con los elementos* <code> *y* <pre>

El Listado 2-24 define un elemento **<article>** que muestra código programado en JavaScript. Debido a que este código no se encuentra dentro de un elemento **<script>**, el navegador no lo ejecuta y, debido a que se incluye dentro de un elemento **<pre>**, se presenta con los saltos de línea y los espacios declarados en el documento.

```
function cambiarColor() {
  document.body.style.backgroundColor = "#0000FF";
}
document.addEventListener("click", cambiarColor);
```

Figura 2-11: *Código de programación presentado con los elementos* <code> *y* <pre>

Lo básico: a veces los nombres de los elementos y su contenido no ofrecen suficiente información al desarrollador para entender el propósito del código. En estas situaciones, HTML nos permite escribir comentarios. Los comentarios son textos que se incluyen en el documento, pero que no son procesados por el navegador. Para agregar un comentario, tenemos que escribir el texto entre las etiquetas **<!--** y **-->**, como en **<!-- Este es un comentario -->**.

Enlaces

Conectar documentos con otros documentos mediante enlaces es lo que hace posible la Web. Como mencionamos anteriormente, un enlace es contenido asociado a una URL que indica la ubicación de un recurso. Cuando el usuario hace clic en el contenido (texto o imagen), el navegador descarga el recurso. HTML incluye el siguiente elemento para crear enlaces.

<a>—Este elemento crea un enlace. El texto o la imagen que representa el enlace se incluye entre las etiquetas de apertura y cierre. El elemento incluye el atributo **href** para especificar la URL del enlace.

Los enlaces se pueden crear para conectar el documento actual con otros documentos en el mismo sitio web o en otros sitios. Por ejemplo, podemos enlazar las opciones en el menú de nuestra página web a otros documentos en nuestro servidor.

```
<nav>
  <a href="index.html">Principal</a> |
  <a href="fotos.html">Fotos</a> |
  <a href="videos.html">Videos</a> |
  <a href="contacto.html">Contacto</a>
</nav>
```

Listado 2-25: *Enlazando el documento a otros documentos con el elemento* <a>

El elemento **<nav>** en el Listado 2-25 incluye cuatro elementos **<a>** por cada opción del menú. Los elementos incluyen el atributo **href** para indicar al navegador el documento que tiene que abrir cuando el usuario hace clic en el enlace. Por defecto, los enlaces se muestran subrayados y en color azul (o violeta si el usuario ya ha hecho clic en ellos).

Principal | Fotos | Videos | Contacto

Figura 2-12: *Hipervínculos*

Cuando el usuario hace clic en cualquiera de estos enlaces, el navegador descarga el documento indicado por el atributo `href` y muestra su contenido en pantalla. Por ejemplo, si hacemos clic en el enlace creado para la opción **Contacto** en el código del Listado 2-25, el navegador descarga el archivo contacto.html y muestra su contenido en la pantalla, como ilustra la Figura 2-13 (este ejemplo asume que hemos creado un archivo llamado contacto.html).

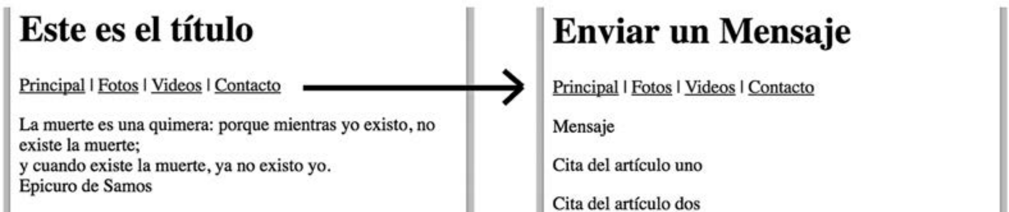

Figura 2-13: *Navegando entre documentos*

Hágalo usted mismo: cree un nuevo archivo llamado contacto.html. Copie el código HTML del archivo index.html en el archivo contacto.html. Reemplace el elemento **<nav>** en ambos archivos con el código del Listado 2-25. Cambie el título en el elemento **<header>** del archivo contacto.html por el texto «Enviar un Mensaje». Abra el archivo index.html en su navegador y haga clic en la opción **Contacto**. El navegador debería abrir el archivo contacto.html y mostrarlo en pantalla, según ilustra la Figura 2-13.

Los documentos enlazados en el menú del Listado 2-25 pertenecen al mismo sitio web y esa es la razón por la que usamos URL relativas para especificar su ubicación, pero si lo que necesitamos es crear enlaces a documentos que no están almacenados en nuestro servidor, tenemos que usar URL absolutas, como en el siguiente ejemplo.

```
<footer>
  <address>Toronto, Canada</address>
  <small>&copy; 2016 <a href="http://www.jdgauchat.com">J.D
Gauchat</a></small>
</footer>
```

Listado 2-26: *Enlazando el documento a documentos en otros sitios web con el elemento* <a>

El código en el Listado 2-26 agrega un enlace al pie de página de nuestro ejemplo que apunta al sitio web www.jdgauchat.com. El enlace trabaja como cualquier otro, pero ahora el navegador tiene la URL completa para acceder al documento (en este caso, el archivo index del sitio web con el domino www.jdgauchat.com).

Toronto, Canada
© 2016 J.D Gauchat

Figura 2-14: *Enlace a un documento externo*

El elemento **<a>** puede incluir el atributo **target** para especificar el destino en el cual el documento será abierto. El valor **_self** se asigna por defecto, lo que significa que el documento se abre en la misma ubicación que el documento actual (el mismo recuadro o ventana). Otros valores son **_blank** (el documento se abre en una nueva ventana), **_parent** (el documento se abre en el recuadro padre), y **_top** (el documento se abre en la ventana actual).

Como veremos más adelante, los documentos se pueden abrir en recuadros insertados dentro de otros documentos. El valor del atributo **target** considera esta jerarquía de recuadros, pero debido a que los recuadros no se usan de forma frecuente en sitios web modernos, los dos valores más comunes son **_self**, para abrir el documento en la misma ventana, y **_blank**, para abrir el documento en una nueva ventana. El siguiente ejemplo implementa el último valor para acceder al dominio www.jdgauchat.com desde una nueva ventana, de modo que el usuario nunca abandona nuestro sitio web.

```
<footer>
  <address>Toronto, Canada</address>
  <small>&copy; 2016 <a href="http://www.jdgauchat.com"
target="_blank">J.D Gauchat</a></small>
</footer>
```

Listado 2-27: *Abriendo un enlace en una nueva ventana*

Además de conectar un documento con otro, los enlaces también se pueden crear hacia otros elementos dentro del mismo documento. Esto es particularmente útil cuando el documento genera una página extensa que el usuario debe desplazar para poder ver todo su contenido. Aprovechando esta característica, podemos crear enlaces hacia diferentes partes de una página. Cuando el usuario quiere ver algo que no es visible al momento, puede hacer clic en estos enlaces y el navegador desplaza la página hasta que el elemento apuntado por el enlace aparece en la pantalla. El elemento que queremos enlazar tiene que ser identificado con el atributo **id**. Para crear un enlace a un elemento, debemos incluir el valor asignado a este atributo precedido por el carácter #, igual que ilustra el siguiente ejemplo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <header id="titulo">
    Este es el título
  </header>
  <nav>
    Principal | Fotos | Videos | Contacto
  </nav>
  <main>
    <section>
      <p>Artículo 1</p>
```

```
      <p>Artículo 2</p>
      <p>Artículo 3</p>
      <p>Artículo 4</p>
      <p><a href="#titulo">Volver</a></p>
    </section>
  </main>
  <footer>
    &copy; Derechos Reservados 2016
  </footer>
</body>
</html>
```

Listado 2-28: *Creando enlaces a elementos en el mismo documento*

En el documento del Listado 2-28, usamos el atributo **id** con el valor "titulo" para identificar el elemento **<header>**. Usando este valor y el carácter #, creamos un enlace al final del contenido que lleva al usuario hacia la parte superior de la página. Cuando el usuario hace clic en el enlace, en lugar de abrir un documento, el navegador desplaza la página hasta que el contenido del elemento **<header>** se vuelve visible.

Hágalo usted mismo: actualice el código en su archivo index.html con el código del Listado 2-28. Agregue más párrafos con elementos **<p>** para generar más contenido dentro del elemento **<section>**. Abra el documento en su navegador, desplace la página hacia abajo, y haga clic en el enlace **Volver**. Si la cabecera no es visible, el navegador desplazará la página hacia arriba para mostrarla en pantalla.

El elemento **<a>** también se puede usar para crear enlaces a aplicaciones. HTML ofrece las palabras clave **mailto** y **tel** para especificar una cuenta de correo o un número de teléfono. Cuando se hace clic en un enlace de estas características, el sistema abre el programa encargado de responder a este tipo de solicitudes (enviar un correo o hacer una llamada telefónica) y le envía los datos especificados en el enlace. El siguiente ejemplo implementa la palabra clave **mailto** para enviar un correo electrónico.

```
<footer>
  <address>Toronto, Canada</address>
  <small>&copy; 2016 <a href="mailto:info@jdgauchat.com">J.D
Gauchat</a></small>
</footer>
```

Listado 2-29: *Enviando un correo electrónico*

Hágalo usted mismo: reemplace el elemento **<footer>** en su archivo index.html con el código del Listado 2-29. Abra el documento en su navegador y haga clic en el enlace. El sistema debería abrir su programa de correo para enviar un mensaje a la cuenta info@jdgauchat.com.

Los documentos a los que se accede a través de un enlace creado por el elemento **<a>** son descargados por el navegador y mostrados en pantalla, pero a veces los usuarios no necesitan

que el navegador abra el documento, sino que el archivo se almacene en sus discos duros para usarlo más adelante. HTML ofrece dos atributos para este propósito:

download—Este es un atributo booleano que, cuando se incluye, indica que en lugar de leer el archivo el navegador debería descargarlo.

ping—Este atributo declara la ruta del archivo que se debe abrir en el servidor cuando el usuario hace clic en el enlace. El valor puede ser una o más URL separadas por un espacio.

Cuando el atributo **download** se encuentra presente dentro de un elemento **<a>**, el archivo especificado por el atributo **href** se descarga y almacena en el disco duro del usuario. Por otro lado, el archivo que indica el atributo **ping** no se descarga, sino que se ejecuta en el servidor. Este archivo se puede usar para ejecutar código que almacena información en el servidor cada vez que el archivo principal se descarga o para llevar un control de las veces que esta acción ocurre. En el siguiente ejemplo, implementamos ambos atributos para permitir al usuario descargar un archivo PDF.

```
<article>
  <p>La muerte es una quimera: porque mientras yo existo, no existe la
muerte;<br>y cuando existe la muerte, ya no existo yo.<br>Epicuro de
Samos</p>
  <footer>
    <a href="http://www.formasterminds.com/content/miarchivo.pdf"
ping="http://www.formasterminds.com/control.php" download>Clic aquí
para descargar</a>
  </footer>
</article>
```

Listado 2-30: *Aplicando los atributos* `ping` *y* `download`

En el ejemplo del Listado 2-30, agregamos un pie de página al artículo de ejemplos anteriores con un enlace a un archivo PDF. En circunstancias normales, un navegador moderno mostraría el contenido del archivo en pantalla, pero en este caso el atributo **download** obliga al navegador a descargar el archivo y almacenarlo en el disco duro.

Este ejemplo incluye un atributo **ping** que apunta a un archivo llamado control.php. Como resultado, cada vez que el usuario hace clic en el enlace, se descarga el archivo PDF y el código PHP se ejecuta en el servidor, lo que permite al desarrollador hacer un seguimiento de las veces que esta acción ocurre (almacenando información acerca del usuario en una base de datos, por ejemplo).

Hágalo usted mismo: reemplace el elemento **<article>** en su archivo index.html por el código del Listado 2-30 y abra el documento en su navegador. Cuando haga clic en el enlace, el navegador debería descargar el archivo PDF. Elimine el atributo **download** para comparar el comportamiento del navegador.

IMPORTANTE: el propósito del atributo **ping** es ejecutar código en el servidor. En el ejemplo hemos usado un archivo con código PHP, pero podría hacer lo mismo con cualquier otro lenguaje de programación que funcione en el servidor, como Python o Ruby. El modo de programar el archivo

asignado al atributo **`ping`** depende del lenguaje que usemos y lo que queramos lograr. El tema va más allá del propósito de este libro. Para obtener más información sobre PHP, visite nuestro sitio web y siga los enlaces de este capítulo.

Imágenes

Las imágenes pueden se pueden considerar el segundo medio más importante en la Web. HTML incluye los siguientes elementos para introducir imágenes en nuestros documentos.

<img>—Este elemento inserta una imagen en el documento. El elemento requiere del atributo **`src`** para especificar la URL del archivo con la imagen que queremos incorporar.

<picture>—Este elemento inserta una imagen en el documento. Trabaja junto con el elemento **`<source>`** para ofrecer múltiples imágenes en diferentes resoluciones. Es útil para crear sitios web adaptables, como veremos en el Capítulo 5.

<figure>—Este elemento representa contenido asociado con el contenido principal, pero que se puede eliminar sin que se vea afectado, como fotos, vídeos, etc.

<figcaption>—Este elemento introduce un título para el elemento **`<figure>`**.

Para incluir una imagen en el documento, solo necesitamos declarar el elemento **`<img>`** y asignar la URL del archivo al atributo **`src`**.

```
<article>
  <p>La muerte es una quimera: porque mientras yo existo, no existe la muerte;<br>y cuando existe la muerte, ya no existo yo.<br>Epicuro de Samos</p>
  <img src="miimagen.jpg">
</article>
```

Listado 2-31: *Incluyendo una imagen en el documento*

El código del Listado 2-31 carga la imagen del archivo miimagen.jpg y la muestra en pantalla en su tamaño original, como lo ilustra la Figura 2-15.

La muerte es una quimera: porque mientras yo existo, no existe la muerte;
y cuando existe la muerte, ya no existo yo.
Epicuro de Samos

Figura 2-15: *Imagen en el documento*

La imagen se representa en su tamaño original, pero podemos definir un tamaño personalizado y algunos parámetros de configuración usando el resto de los atributos disponibles para el elemento `<img>`.

width—Este atributo declara el ancho de la imagen.

height—Este atributo declara la altura de la imagen.

alt—Este atributo especifica el texto que se muestra cuando la imagen no se puede cargar.

srcset—Este atributo nos permite especificar una lista de imágenes de diferentes resoluciones que el navegador puede cargar para el mismo elemento.

sizes—Este atributo especifica una lista de *media queries* (consulta de medios) junto con distintos tamaños de imágenes para que el navegador decida qué mostrar según la resolución de la pantalla. Estudiaremos *media queries* y diseño web adaptable en el Capítulo 5.

crossorigin—Este atributo establece las credenciales para imágenes de origen cruzado (múltiples orígenes). Los valores posible son **anonymous** (sin credenciales) y **use-credentials** (requiere credenciales). Estudiaremos la tecnología CORS y cómo trabajar con imágenes procedentes de diferentes orígenes en el Capítulo 11.

Los atributos **width** y **height** determinan las dimensiones de la imagen, pero no tienen en cuenta la relación. Si declaramos ambos valores sin considerar la proporción original de la imagen, el navegador deberá estirar o achatar la imagen para adaptarla a las dimensiones definidas. Para reducir la imagen sin cambiar la proporción original, podemos especificar uno solo de los atributos y dejar que el navegador calcule el otro.

```
<article>
  <p>La muerte es una quimera: porque mientras yo existo, no existe la
muerte;<br>y cuando existe la muerte, ya no existo yo.<br>Epicuro de
Samos</p>
  <img src="miimagen.jpg" width="150" alt="Arboles en mi patio">
</article>
```

Listado 2-32: *Reduciendo el tamaño de la imagen*

El código en el Listado 2-32 agrega el atributo **width** con el valor 150 al elemento **<img>** introducido en el ejemplo anterior. Esto reduce el ancho de la imagen a 150 píxeles, pero como el atributo **height** no se ha declarado, la altura de la imagen se calcula automáticamente considerando las proporciones originales de la imagen. El resultado se muestra en la Figura 2-16.

Figura 2-16: *Imagen con un tamaño personalizado*

Hágalo usted mismo: descargue la imagen miimagen.jpg desde nuestro sitio web. Reemplace el elemento **<article>** en su archivo index.html por el código del Listado 2-31 y abra el documento en su navegador. Debería ver algo similar a la Figura 2-15. Repita el proceso con el código del Listado 2-32. Ahora debería ver la imagen reducida, según se ilustra en la Figura 2-16. El elemento **<img>** en el código del Listado 2-32 también incluye el atributo **alt**. Para probar este atributo, borre el archivo miimagen.jpg y actualice el documento en su navegador. Como el navegador ya no puede encontrar el archivo de la imagen, mostrará el texto asignado al atributo **alt** en su lugar.

Algunas imágenes, como los iconos, son importantes porque otorgan un significado al resto del contenido, pero otras, como la imagen de estos ejemplos, actúan como complemento y se pueden eliminar sin que afecten al flujo de información. Cuando esta clase de información se encuentra presente, se puede utilizar el elemento **<figure>** para identificarla. Este elemento se suele implementar junto con el elemento **<figcaption>** para incluir texto descriptivo. En el siguiente ejemplo usamos estos dos elementos para identificar nuestra imagen y mostrar su título al usuario.

```
<article>
  <p>La muerte es una quimera: porque mientras yo existo, no existe la
muerte;<br>y cuando existe la muerte, ya no existo yo.<br>Epicuro de
Samos</p>
  <figure>
    <img src="miimagen.jpg" width="150" alt="Arboles en mi patio">
    <figcaption>Arboles en mi patio<figcaption>
  </figure>
</article>
```

Listado 2-33: *Identificando la imagen con el elemento* <figure>

Por defecto, los navegadores asignan márgenes laterales al elemento **<figure>**. El resultado se muestra en la Figura 2-17.

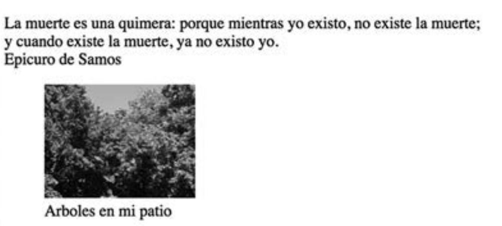

Figura 2-17: *Imagen junto a su título*

Listados

A menudo, la información se debe representar como una lista de ítems. Por ejemplo, muchos sitios web incluyen listados de libros, películas, o términos y descripciones. Para crear estos listados, HTML ofrece los siguientes elementos.

**
—Este elemento crea una lista de ítems sin orden. Está compuesto por etiquetas de apertura y cierre para agrupar los ítems (
** y **
) y trabaja junto con el elemento **<li> para definir cada uno de los ítems de la lista.

**
—Este elemento crea una lista ordenada de ítems. Está compuesto por etiquetas de apertura y cierre para agrupar los ítems (
** y **
) y trabaja junto con el elemento **<li> para definir los ítems de la lista. Este elemento puede incluir los atributos **reversed** para invertir el orden de los indicadores, **start** para determinar el valor desde el cual los indicadores tienen que comenzar a contar y **type** para determinar el tipo de indicador que queremos usar. Los valores disponibles para el atributo **type** son **1** (números), **a** (letras minúsculas), **A** (letras mayúsculas), **i** (números romanos en minúsculas) e **I** (números romanos en mayúsculas).

<dl>—Este elemento crea una lista de términos y descripciones. El elemento trabaja junto con los elementos **<dt>** y **<dd>** para definir los ítems de la lista. El elemento **<dl>** define la lista, el elemento **<dt>** define los términos y el elemento **<dd>** define las descripciones.

El elemento que usamos para crear la lista depende de las características del contenido. Por ejemplo, si el orden de los ítems no es importante, podemos usar el elemento **<ul>**. En esta clase de listas, los ítems se declaran entre las etiquetas **<ul>** con el elemento **<li>**, como muestra el siguiente ejemplo.

```
<aside>
  <ul>
    <li>IT, Stephen King</li>
    <li>Carrie, Stephen King</li>
    <li>El Resplandor, Stephen King</li>
    <li>Misery, Stephen King</li>
  </ul>
</aside>
```

Listado 2-34: *Creando una lista de ítems sin orden*

El elemento **<ul>** presenta los ítems en el orden en el que se han declarado en el código y los identifica con un punto del lado izquierdo.

- IT, Stephen King
- Carrie, Stephen King
- El Resplandor, Stephen King
- Misery, Stephen King

Figura 2-18: *Lista sin orden*

Si necesitamos declarar la posición de cada ítem, podemos crear la lista con el elemento **<ol>**. Este elemento crea una lista de ítems en el orden en el que se han declarado en el código, pero en lugar de usar puntos para identificarlos, les asigna un valor. Por defecto, los indicadores se crean con números, pero podemos cambiarlos con el atributo **type**. En el siguiente ejemplo, usamos letras mayúsculas.

```
<aside>
  <ol type="A">
    <li>IT, Stephen King</li>
    <li>Carrie, Stephen King</li>
    <li>El Resplandor, Stephen King</li>
    <li>Misery, Stephen King</li>
  </ol>
</aside>
```

Listado 2-35: *Creando una lista ordenada de ítems*

A. IT, Stephen King
B. Carrie, Stephen King
C. El Resplandor, Stephen King
D. Misery, Stephen King

Figura 2-19: *Lista ordenada*

Aunque los ítems se muestran siempre en el orden en el que se han declarado en el código, podemos utilizar otros atributos del elemento **<ol>** para cambiar el orden de los indicadores. Por ejemplo, agregando el atributo **reversed** logramos que los indicadores se muestren en orden invertido.

```
<aside>
  <ol type="A" reversed>
    <li>IT, Stephen King</li>
    <li>Carrie, Stephen King</li>
    <li>El Resplandor, Stephen King</li>
    <li>Misery, Stephen King</li>
  </ol>
</aside>
```

Listado 2-36: *Creando una lista en orden invertido*

D. IT, Stephen King
C. Carrie, Stephen King
B. El Resplandor, Stephen King
A. Misery, Stephen King

Figura 2-20: *Lista en orden invertido*

Los elementos **<dl>**, **<dt>**, y **<dd>** trabajan de forma similar al elemento **<ul>**, pero su propósito es mostrar una lista de términos y descripciones. Los términos se representan por el elemento **<dt>** y las descripciones por el elemento **<dd>**. En el siguiente ejemplo, los usamos para agregar la descripción de los libros listados anteriormente.

```
<aside>
  <dl>
    <dt>IT</dt>
```

```
    <dd>Tras lustros de tranquilidad y lejania una antigua promesa
infantil les hace volver al lugar en el que vivieron su infancia y
juventud como una terrible pesadilla.</dd>
    <dt>Carrie</dt>
    <dd>Sus compañeros se burlan de ella, pero Carrie tiene un don.
Puede mover cosas con su mente. Este es su poder y su gran
problema.</dd>
    <dt>El Resplandor</dt>
    <dd>Al escritor Jack Torrance le es ofrecido un empleo como cuidador
del hotel Overlook durante el invierno junto a su familia.</dd>
    <dt>Misery</dt>
    <dd>Paul Sheldon es un famoso escritor de novelas románticas
ambientadas en la época victoriana, cuyo personaje principal se llama
Misery Chastain.</dd>
  </dl>
</aside>
```

Listado 2-37: *Creando una lista de términos y descripciones*

Por defecto, los navegadores otorgan estilos a este tipo de listas que incluyen márgenes a los lados para diferenciar los términos de las descripciones. El resultado se muestra en la Figura 2-21.

IT
    Tras lustros de tranquilidad y lejania una antigua promesa infantil les hace volver al lugar en el que vivieron su infancia y juventud como una terrible pesadilla.
Carrie
    Sus compañeros se burlan de ella, pero Carrie tiene un don. Puede mover cosas con su mente. Este es su poder y su gran problema.
El Resplandor
    Al escritor Jack Torrance le es ofrecido un empleo como cuidador del hotel Overlook durante el invierno junto a su familia.
Misery
    Paul Sheldon es un famoso escritor de novelas románticas ambientadas en la época victoriana, cuyo personaje principal se llama Misery Chastain.

Figura 2-21: *Lista de términos y descripciones*

Hágalo usted mismo: reemplace el elemento **`<aside>`** en su archivo index.html con el ejemplo que quiere probar y abra el documento en su navegador. Inserte el atributo **`start`** con el valor "3" en el elemento **`<ol>`** del Listado 2-35. Los indicadores deberían comenzar a contar desde la letra C.

Los siguientes elementos se han diseñado con propósitos diferentes, pero también se utilizan frecuentemente para construir listas de ítems.

<blockquote>—Este elemento representa un bloque de texto que incluye una cita tomada de otro texto en el documento.

<details>—Este elemento crea una herramienta que se expande cuando se hace clic en ella para mostrar información adicional. La parte visible se define con el elemento **`<summary>`**, y se pueden usar elementos comunes como **`<p>`** para definir el contenido.

El elemento **`<blockquote>`** es similar al elemento **`<p>`**, pero como también incluye márgenes a los costados, se ha usado tradicionalmente para presentar listas de valores, como lo demuestra el siguiente ejemplo.

```
<aside>
  <blockquote>IT</blockquote>
  <blockquote>Carrie</blockquote>
  <blockquote>El Resplandor</blockquote>
  <blockquote>Misery</blockquote>
</aside>
```

Listado 2-38: *Creando una lista con el elemento* `<blockquote>`

El listado generado por el elemento **<blockquote>** se muestra igual que otras listas, pero no incluye puntos o números para identificar cada ítem.

IT

Carrie

El Resplandor

Misery

Figura 2-22: *Listado de ítems con el elemento* `<blockquote>`

En sitios web modernos son comunes las herramientas que revelan información adicional cuando el usuario lo requiere. Para ofrecer esta posibilidad, HTML incluye el elemento **<details>**. Este elemento muestra un título, especificado por el elemento **<summary>**, y contenido que se puede representar por elementos comunes como **<p>** o **<blockquote>**. Debido a esto, el elemento **<details>** se puede utilizar para revelar una lista de valores, como lo hacemos en el siguiente ejemplo.

```
<details>
  <summary>My Books</summary>
  <p>IT</p>
  <p>Carrie</p>
  <p>El Resplandor</p>
  <p>Misery</p>
</details>
```

Listado 2-39: *Revelando información con los elementos* `<details>` *y* `<summary>`

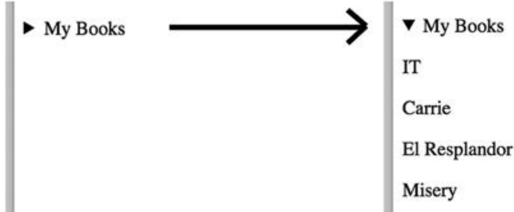

Figura 2-23: *El elemento* `<details>` *antes y después de ser cliqueado*

Tablas

Las tablas organizan información en filas y columnas. Debido a sus características, se usaron durante mucho tiempo para estructurar documentos HTML, pero con la introducción de CSS, los desarrolladores pudieron lograr el mismo efecto implementando otros elementos. Aunque ya no se recomienda usar tablas para definir la estructura de un documento, todavía se utilizan para presentar información tabular, como estadísticas o especificaciones técnicas, por ejemplo. HTML incluye varios elementos para crear una tabla. Los siguientes son los más utilizados.

<table>—Este elemento define una tabla. Incluye etiquetas de apertura y cierre para agrupar el resto de los elementos que definen la tabla.

<tr>—Este elemento define una fila de celdas. Incluye etiquetas de apertura y cierre para agrupar las celdas.

<td>—Este elemento define una celda. Incluye etiquetas de apertura y cierre para delimitar el contenido de la celda y puede incluir los atributos **colspan** y **rowspan** para indicar cuántas columnas y filas ocupa la celda.

<th>—Este elemento define una celda para la cabecera de la tabla. Incluye etiquetas de apertura y cierre para delimitar el contenido de la celda y puede incluir los atributos **colspan** y **rowspan** para indicar cuántas columnas y filas ocupa la celda.

Para incluir una tabla en el documento, primero tenemos que declarar el elemento **<table>** y luego describir las filas una por una con los elementos **<tr>** y **<td>**, como muestra el siguiente ejemplo.

```
<article>
  <table>
    <tr>
      <td>IT</td>
      <td>Stephen King</td>
      <td>1986</td>
    </tr>
    <tr>
      <td>Carrie</td>
      <td>Stephen King</td>
      <td>1974</td>
    </tr>
    <tr>
      <td>El Resplandor</td>
      <td>Stephen King</td>
      <td>1977</td>
    </tr>
  </table>
</article>
```

Listado 2-40: *Creando una tabla*

Debido a que el navegador interpreta el documento de forma secuencial desde la parte superior a la inferior, cada vez que declaramos una fila, tenemos que declarar las celdas que

corresponden a esa fila y su contenido. Siguiendo este patrón, en el Listado 2-40, creamos una tabla para mostrar libros, uno por fila. La primer celda de cada fila representa el título del libro, la segunda celda representa el autor, y la tercera celda el año de publicación. Cuando el navegador abre este documento, muestra la información en el orden en el que se ha declarado en el código y con el tamaño determinado por el contenido de las celdas.

IT	Stephen King	1986
Carrie	Stephen King	1974
El Resplandor	Stephen King	1977

Figura 2-24: *Tabla con estilos por defecto*

Si queremos incluir una cabecera para describir el contenido de cada columna, podemos crear una fila de celdas adicional representadas con el elemento **<th>**.

```
<article>
  <table>
    <tr>
      <th>Título</th>
      <th>Autor</th>
      <th>Año</th>
    </tr>
    <tr>
      <td>IT</td>
      <td>Stephen King</td>
      <td>1986</td>
    </tr>
    <tr>
      <td>Carrie</td>
      <td>Stephen King</td>
      <td>1974</td>
    </tr>
    <tr>
      <td>El Resplandor</td>
      <td>Stephen King</td>
      <td>1977</td>
    </tr>
  </table>
</article>
```

Listado 2-41: *Creando una tabla con cabecera*

Por defecto, los navegadores muestran las cabeceras con el texto en negrita y centrado.

Título	Autor	Año
IT	Stephen King	1986
Carrie	Stephen King	1974
El Resplandor	Stephen King	1977

Figura 2-25: *Cabecera de tabla con estilos por defecto*

Las celdas se pueden estirar para que ocupen más de una columna con los atributos **colspan** y **rowspan**. Por ejemplo, podemos usar solo una celda de cabecera para identificar el título y el autor del libro.

```
<article>
  <table>
    <tr>
      <th colspan="2">Libro</th>
      <th>Año</th>
    </tr>
    <tr>
      <td>IT</td>
      <td>Stephen King</td>
      <td>1986</td>
    </tr>
    <tr>
      <td>Carrie</td>
      <td>Stephen King</td>
      <td>1974</td>
    </tr>
    <tr>
      <td>El Resplandor</td>
      <td>Stephen King</td>
      <td>1977</td>
    </tr>
  </table>
</article>
```

Listado 2-42: *Estirando celdas*

El ejemplo del Listado 2-42 incluye una celda de cabecera con el título **Libro** para las primeras dos columnas. Debido al valor asignado al atributo **colspan**, esta celda se estira para que ocupe el espacio de dos. El resultado se muestra en la Figura 2-26.

Libro		Año
IT	Stephen King	1986
Carrie	Stephen King	1974
El Resplandor	Stephen King	1977

Figura 2-26: *Celdas de múltiples columnas*

Lo básico: HTML también incluye los elementos **<thead>**, **<tbody>**, y **<tfoot>** para representar la cabecera, el cuerpo, y el pie de la tabla, respectivamente, y otros elementos como **<colgroup>** para agrupar columnas. Para obtener más información, visite nuestro sitio web y siga los enlaces de este capítulo.

Atributos Globales

La mayoría de los navegadores actuales automáticamente traducen el contenido del documento cuando detectan que se ha escrito en un idioma diferente al del usuario, pero en

algunos casos la página puede incluir frases o párrafos enteros que no se deben alterar, como nombres de personas o títulos de películas. Para controlar el proceso de traducción, HTML ofrece un atributo global llamado **`translate`**. Este atributo puede tomar dos valores: **`yes`** y **`no`**. Por defecto, el valor es **`yes`** (si). En el siguiente ejemplo, usamos un elemento **`<span>`** para especificar la parte del texto que no se debería traducir.

```
<p>My favorite movie is <span translate="no">Two Roads</span></p>
```

Listado 2-43: *Usando el atributo* `translate`

El elemento **`<span>`** se diseñó para presentar texto, pero a diferencia del elemento **`<p>`** no asigna ningún estilo al contenido, por lo que el texto se presenta en la misma línea y con el tipo de letra y tamaño por defecto. La Figura 2-27 muestra lo que vemos en la ventana del navegador después de traducirlo.

Mi película favorita es Two Roads

Figura 2-27: *texto traducido por el navegador*

Hágalo usted mismo: inserte el código del Listado 2-43 en la sección principal de su documento y abra el documento en su navegador. Los navegadores como Google Chrome ofrecen una opción para traducir el documento en un menú contextual cuando hacemos clic con el botón derecho del ratón. Si la traducción no se realiza de forma automática, seleccione esta opción para traducir el texto. Una vez traducido el texto, debería ver algo similar a lo que se muestra en la Figura 2-27.

Otro atributo útil que podemos agregar a un elemento HTML es **`contenteditable`**. Este es un atributo booleano que, si está presente, permite al usuario editar el contenido del elemento. Si el usuario hace clic en un elemento que contiene este atributo, puede cambiar su contenido. El siguiente ejemplo implementa el elemento **`<span>`** nuevamente para permitir a los usuarios editar el nombre de una película.

```
<p>Mi película favorita es <span contenteditable>Casablanca</span></p>
```

Listado 2-44: *Usando el atributo* `contenteditable` *para editar contenido*

Hágalo usted mismo: reemplace el párrafo del Listado 2-43 por el párrafo del Listado 2-44 y abra el documento en su navegador. Haga clic en el nombre de la película para cambiarlo.

IMPORTANTE: las modificaciones introducidas por el usuario solo se encuentran disponibles en su ordenador. Si queremos que los cambios se almacenen en el servidor, tenemos que subir esta información con un programa en JavaScript usando Ajax, según veremos en el Capítulo 21.

2.3 Formularios

Los formularios son herramientas que podemos incluir en un documento para permitir a los usuarios insertar información, tomar decisiones, comunicar datos y cambiar el comportamiento de una aplicación. El propósito principal de los formularios es permitir al usuario seleccionar o insertar información y enviarla al servidor para ser procesada. La Figura 2-28 muestra algunas de las herramientas facilitadas este fin.

Figura 2-28: *Formulario en el navegador*

Definición

Como se muestra en la Figura 2-28, los formularios pueden presentar varias herramientas que permiten al usuario interactuar con el documento, incluidos campos de texto, casillas de control, menús desplegables y botones. Cada una de estas herramientas se representa por un elemento y el formulario queda definido por el elemento `<form>`, que incluye etiquetas de apertura y cierre para agrupar al resto de los elementos y requiere de algunos atributos para determinar cómo se envía la información al servidor.

name—Este atributo especifica el nombre del formulario. También se encuentra disponible para otros elementos, pero es particularmente útil para elementos de formulario, como veremos más adelante.

method—Este atributo determina el método a utilizar para enviar la información al servidor. Existen dos valores disponibles: `GET` y `POST`. El método `GET` se usa para enviar una cantidad limitada de información de forma pública (los datos son incluidos en la URL, la cual no puede contener más de 255 caracteres). Por otro lado, el método `POST` se utiliza para enviar una cantidad ilimitada de información de forma privada (los datos no son visibles al usuario y pueden tener la longitud que necesitemos).

action—Este atributo declara la URL del archivo en el servidor que va a procesar la información enviada por el formulario.

target—Este atributo determina dónde se mostrará la respuesta recibida desde el servidor. Los valores disponibles son `_blank` (nueva ventana), `_self` (mismo recuadro), `_parent` (recuadro padre), y `_top` (la ventana que contiene el recuadro). El valor `_self` se declara por defecto, lo que significa que la respuesta recibida desde el servidor se mostrará en la misma ventana.

enctype—Este atributo declara la codificación aplicada a los datos que envía el formulario. Puede tomar tres valores: `application/x-www-form-urlencoded` (los caracteres son codificados), `multipart/form-data` (los caracteres no son codificados), `text/plain` (solo los espacios son codificados). El primer valor se asigna por defecto.

accept-charset—Este atributo declara el tipo de codificación aplicada al texto del formulario. Los valores más comunes son **UTF-8** e **ISO-8859-1**. El valor por defecto se asigna al documento con el elemento **<meta>** (ver Listado 2-6).

El siguiente ejemplo define un formulario básico. El atributo **name** identifica el formulario con el nombre "formulario", el atributo **method** determina que los datos se incluirán en la URL (GET), y el atributo **action** declara que procesar.php es el archivo que se ejecutará en el servidor para procesar la información y devolver el resultado.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">

    </form>
  </section>
</body>
</html>
```

Listado 2-45: *Definiendo un formulario con el elemento* <form>

Hágalo usted mismo: cree un nuevo archivo HTML en su editor o modifique el archivo del ejemplo anterior con el código del Listado 2-45 y abra el documento en su navegador. En este momento no verá nada en la pantalla porque el formulario se ha declarado vacío. A continuación, desarrollaremos su contenido.

Elementos

Un formulario puede incluir diferentes herramientas para permitir al usuario seleccionar o insertar información. HTML incluye múltiples elementos para crear estas herramientas. Los siguientes son los más utilizados.

<input>—Este elemento crea un campo de entrada. Puede recibir diferentes tipos de entradas, dependiendo del valor del atributo **type**.

<textarea>—Este elemento crea un campo de entrada para insertar múltiples líneas de texto. El tamaño se puede declarar en números enteros usando los atributos **rows** y **cols**, o en píxeles con estilos CSS, como veremos en el Capítulo 3.

<select>—Este elemento crea una lista de opciones que el usuario puede elegir. Trabaja junto con el elemento **<option>** para definir cada opción y el elemento **<optgroup>** para organizar las opciones en grupos.

<button>—Este elemento crea un botón. Incluye el atributo **`type`** para definir el propósito del botón. Los valores disponibles son **`submit`** para enviar el formulario (por defecto), **`reset`** para reiniciar el formulario, y **`button`** para realizar tareas personalizadas.

<output>—Este elemento representa un resultado producido por el formulario. Se implementa por medio de código JavaScript para mostrar el resultado de una operación al usuario.

<meter>—Este elemento representa una medida o el valor actual de un rango.

<progress>—Este elemento representa el progreso de una operación.

<datalist>—Este elemento crea un listado de valores disponibles para otros controles. Trabaja junto con el elemento **`<option>`** para definir cada valor.

<label>—Este elemento crea una etiqueta para identificar un elemento de formulario.

<fieldset>—Este elemento agrupa otros elementos de formulario. Se usa para crear secciones dentro de formularios extensos. El elemento puede contener un elemento **`<legend>`** para definir el título de la sección.

El elemento **`<input>`** es el más versátil de todos. Este elemento genera un campo de entrada en el que el usuario puede seleccionar o insertar información, pero puede adoptar diferentes características y aceptar varios tipos de valores dependiendo del valor de su atributo **`type`**. Los siguientes son los valores disponibles para este atributo.

text—Este valor genera un campo de entrada para insertar texto genérico.

email—Este valor genera un campo de entrada para insertar cuentas de correo.

search—Este valor genera un campo de entrada para insertar términos de búsqueda.

url—Este valor genera un campo de entrada para insertar URL.

tel—Este valor genera un campo de entrada para insertar números de teléfono.

number—Este valor genera un campo de entrada para insertar números.

range—Este valor genera un campo de entrada para insertar un rango de números.

date—Este valor genera un campo de entrada para insertar una fecha.

datetime-local—Este valor genera un campo de entrada para insertar fecha y hora.

week—Este valor genera un campo de entrada para insertar el número de la semana (dentro del año).

month—Este valor genera un campo de entrada para insertar el número del mes.

time—Este valor genera un campo de entrada para insertar una hora (horas y minutos).

hidden—Este valor oculta el campo de entrada. Se usa para enviar información complementaria al servidor.

password—Este valor genera un campo de entrada para insertar una clave. Reemplaza los caracteres insertados con estrellas o puntos para ocultar información sensible.

color—Este valor genera un campo de entrada para insertar un color.

checkbox—Este valor genera una casilla de control que permite al usuario activar o desactivar una opción.

radio—Este valor genera un botón de opción para seleccionar una opción de varias posibles.

file—Este valor genera un campo de entrada para seleccionar un archivo en el ordenador del usuario.

button—Este valor genera un botón. El botón trabaja como el elemento **<button>** de tipo **button**. No realiza ninguna acción por defecto; la acción debe ser definida desde JavaScript, como veremos en próximos capítulos.

submit—Este valor genera un botón para enviar el formulario.

reset—Este valor genera un botón para reiniciar el formulario.

image—Este valor carga una imagen que se usa como botón para enviar el formulario. Un elemento **<input>** de este tipo debe incluir el atributo **src** para especificar la URL de la imagen.

Para incluir un formulario en nuestro documento, tenemos que declararlo con el elemento **<form>**, como hemos hecho en el ejemplo anterior, y luego incorporar en su interior todos los elementos que el usuario necesitará para insertar la información y enviarla al servidor. Por ejemplo, si queremos que el usuario inserte su nombre y edad, tenemos que incluir dos campos de entrada para texto y un tercer elemento para crear el botón con el que enviar el formulario.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p><input type="text" name="nombre"></p>
      <p><input type="text" name="edad"></p>
      <p><input type="submit"></p>
    </form>
  </section>
</body>
</html>
```

Listado 2-46: *Incluyendo herramientas en un formulario*

La información insertada en el formulario se envía al servidor para ser procesada. Para que el servidor pueda identificar cada valor, los elementos deben incluir el atributo **name**. Con este atributo podemos asignar un nombre único a cada elemento. En el ejemplo del Listado 2-46, llamamos a los campos de entrada "nombre" y "edad" (el elemento **<input>** que crea el botón para enviar el formulario no necesita un nombre porque no envía ningún dato al servidor).

Los elementos de formulario no generan un salto de línea; se muestran en la pantalla uno detrás del otro. Si queremos que el navegador muestre un elemento en cada línea, tenemos que modificar el diseño nosotros mismos. En el Listado 2-46, usamos elementos **<p>** para separar los elementos del formulario, pero este diseño normalmente se logra a través de estilos CSS, como veremos en el Capítulo 4. El resultado se muestra en la Figura 2-29.

Figura 2-29: *Formulario con dos campos de entrada y un botón para enviar los datos*

Otro atributo que podemos usar en este ejemplo es **value**. El tipo de entrada **submit** crea un botón para enviar el formulario. Por defecto, los navegadores le dan al botón el título **Enviar** (Submit), pero podemos usar el atributo **value** para modificarlo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p><input type="text" name="nombre"></p>
      <p><input type="text" name="edad"></p>
      <p><input type="submit" value="Enviar Datos"></p>
    </form>
  </section>
</body>
</html>
```

Listado 2-47: *Asignando un título diferente para el botón Enviar*

El atributo **value** también se puede usar para declarar el valor inicial de un elemento. Por ejemplo, podemos insertar la edad del usuario en el campo **edad** si ya la conocemos.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p><input type="text" name="nombre"></p>
      <p><input type="text" name="edad" value="35"></p>
```

```
      <p><input type="submit" value="Enviar Datos"></p>
    </form>
  </section>
</body>
</html>
```

Listado 2-48: *Declarando valores iniciales*

Hágalo usted mismo: copie el código del Listado 2-48 dentro de su archivo HTML y abra el documento en su navegador. Debería ver un formulario similar al de la Figura 2-29 pero con el número 35 dentro del campo **`edad`** y el botón para enviar el formulario con el título **Enviar datos**.

Los formularios necesitan incluir descripciones que le indiquen al usuario lo datos que debe introducir. Por esta razón, HTML incluye el elemento **`<label>`**. Debido a que estos elementos no solo le indican al usuario qué valor debe introducir, sino que además ayudan al navegador a identificar cada parte del formulario, tienen asociarse al elemento al que están describiendo. Para asociar un elemento **`<label>`** con el elemento de formulario correspondiente, podemos incluir el elemento de formulario dentro del elemento **`<label>`**, como muestra el siguiente ejemplo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p>
        <label>Nombre: <input type="text" name="nombre"></label>
      </p>
      <p>
        <label>Edad: <input type="text" name="edad"></label>
      </p>
      <p><input type="submit" value="Enviar"></p>
    </form>
  </section>
</body>
</html>
```

Listado 2-49: *Identificando elementos de formulario*

Otra alternativa para asociar un elemento **`<label>`** con su elemento de formulario es implementando el atributo **`for`**. El atributo **`for`** conecta el elemento **`<label>`** con el elemento de formulario por medio del valor del atributo **`id`**, según ilustra el siguiente ejemplo.

```
<!DOCTYPE html>
<html lang="es">
<head>
```

```
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p>
        <label for="nombre">Nombre: </label>
        <input type="text" name="nombre" id="nombre"></label>
      </p>
      <p>
        <label for="edad">Edad: </label>
        <input type="text" name="edad" id="edad"></label>
      </p>
      <p><input type="submit" value="Enviar"></p>
    </form>
  </section>
</body>
</html>
```

Listado 2-50: *Asociando etiquetas con elementos por medio del atributo* `for`

El elemento **`<label>`** no incluye ningún estilo por defecto; lo único que hace es asociar una etiqueta con un elemento y, por lo tanto, el texto se muestra en pantalla con el tipo de letra y el tamaño por defecto.

Figura 2-30: *Elementos identificados con una etiqueta*

Los campos de entrada usados en los ejemplos anteriores eran de tipo **`text`**, lo que significa que los usuarios pueden introducir cualquier clase de texto que deseen, pero esto no es lo que necesitamos para este formulario. El primer campo espera un nombre, por lo que no debería permitir que se introduzcan números o textos muy extensos, y el segundo campo espera la edad del usuario, por lo que no debería aceptar ningún tipo de carácter excepto números. Para determinar cuántos caracteres se pueden introducir, el elemento **`<input>`** debe incluir los siguientes atributos.

maxlength—Este atributo especifica el máximo número de caracteres que se permite introducir en el campo.

minlength—Este atributo especifica el mínimo número de caracteres que se permite introducir en el campo.

El siguiente ejemplo limita el nombre a un máximo de 15 caracteres.

```
<!DOCTYPE html>
<html lang="es">
```

```
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p><label>Nombre: <input type="text" name="nombre"
maxlength="15"></label></p>
      <p><label>Edad: <input type="text" name="edad"></label></p>
      <p><input type="submit" value="Enviar"></p>
    </form>
  </section>
</body>
</html>
```

Listado 2-51: *Declarando el máximo número de caracteres permitidos*

El atributo **maxlength** implementado en el formulario del Listado 2-51 limita el número de caracteres que el usuario puede introducir, pero el tipo de campo es aún **text**, lo que significa que en el campo se puede escribir cualquier valor. Si el usuario escribe un número en el campo **nombre** o letras en el campo **edad**, el navegador considerará la entrada válida. Para controlar lo que el usuario puede introducir, tenemos que declarar un tipo de campo diferente con el atributo **type**. El siguiente ejemplo declara el tipo **number** para el campo **edad** para permitir que solo se introduzcan números, e incluye otros campos para que el usuario pueda declarar su cuenta de correo, número de teléfono y sitio web.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p><label>Nombre: <input type="text" name="nombre"
maxlength="15"></label></p>
      <p><label>Edad: <input type="number" name="edad"></label></p>
      <p><label>Correo: <input type="email" name="correo"></label></p>
      <p><label>Teléfono: <input type="tel" name="telefono"></label></p>
      <p><label>Sitio Web: <input type="url" name="sitioweb"></label></p>
      <p><input type="submit" value="Enviar"></p>
    </form>
  </section>
</body>
</html>
```

Listado 2-52: *Solicitando tipos de entrada específicos*

El tipo **number** asignado al campo **edad** en el Listado 2-52 le dice al elemento que solo acepte números. El resto de los tipos de entrada implementados en este ejemplo no imponen ninguna restricción en los caracteres introducidos, pero le indican al navegador la clase de valores que se esperan del usuario. Por ejemplo, el tipo **email** espera una cuenta de correo,

de modo que si el dato introducido no es una cuenta de correo, el navegador no permite que se envíe el formulario y muestra un error en pantalla. El tipo **`url`** trabaja exactamente igual que el tipo **`email`**, pero con direcciones web. Este tipo de campo acepta solo URL absolutas y devuelve un error si el valor no es válido. Otros tipos como **`tel`** no exigen ninguna sintaxis en particular pero le solicitan al navegador que sugiera al usuario posibles valores a introducir o incluya un teclado específico en los dispositivos que lo requieren (en dispositivos móviles, el teclado que se muestra cuando el usuario hace clic en el campo **`telefono`** incluye solo dígitos para facilitar que se introduzcan números telefónicos.

Aunque estos tipos de campo presentan sus propias restricciones, todos se ven iguales en el navegador.

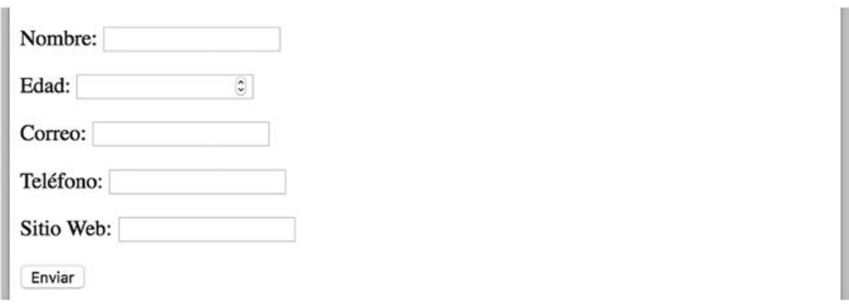

Figura 2-31: *Campos de entrada de diferentes tipos*

Como ilustra la Figura 2-31, por defecto los navegadores incluyen flechas en el lado derecho de un campo de tipo **`number`** con las que podemos seleccionar un número. Para establecer restricciones en los números que el usuario puede seleccionar con estas flechas o controlar los números permitidos, los elementos **`<input>`** de este tipo pueden incluir los siguientes atributos.

min—El valor de este atributo determina el valor mínimo que acepta el campo.

max—El valor de este atributo determina el valor máximo que acepta el campo.

step—El valor de este atributo determina el número por el cual el valor del campo se puede incrementar o reducir. Por ejemplo, si declaramos el valor 5 para este atributo y un valor mínimo de 0 y uno máximo de 10 para el campo, el navegador no nos dejará introducir valores entre 0 y 5 o 5 y 10.

El siguiente ejemplo restringe el valor insertado en el campo **`edad`** de nuestro formulario a un mínimo de 13 y un máximo de 100.

```
<input type="number" name="edad" min="13" max="100">
```

Listado 2-53: *Restringiendo los números*

Otro tipo de entrada que implementa estos mismo atributos es **`range`**. El tipo **`range`** crea un campo que nos permite seleccionar un número desde un rango de valores. El valor inicial se establece de acuerdo a los valores de los atributos **`min`** y **`max`**, pero podemos declarar un valor específico con el atributo **`value`**, como muestra el siguiente ejemplo.

```
<input type="range" name="edad" min="13" max="100" value="35">
```

Listado 2-54: *Implementando el tipo* `range`

Los navegadores muestran un campo de entrada de tipo **`range`** como un control que el usuario puede deslizar para seleccionar un valor.

Figura 2-32: *Campo de entrada de tipo* `range`

Hágalo usted mismo: reemplace el elemento **`<input>`** en su archivo HTML con el elemento de los Listados 2-53 o 2-54 para probar estos ejemplos. Abra el documento en su navegador e inserte o seleccione un número para ver cómo trabajan estos tipos de entrada. Intente enviar el formulario con un valor menor o mayor a los permitidos. El navegador debería mostrarle un error.

Los valores para el tipo **`range`** implementado en el ejemplo anterior no se introducen en un campo de texto, sino que se seleccionan desde una herramienta visual generada por el navegador. El tipo **`range`** no es el único que presenta esta clase de herramientas. Por ejemplo, el tipo **`radio`** crea un botón circular que se resalta cuando se selecciona (ver Figura 2-28). Esto nos permite crear una lista de valores que el usuario puede seleccionar con solo hacer clic en el botón correspondiente. Para ofrecer al usuario todas las opciones disponibles, tenemos que insertar un elemento **`<input>`** por cada opción. Los elementos **`<input>`** se asocian entre ellos por medio del valor del atributo **`name`**, y el valor de cada opción se define por el atributo **`value`**, como muestra el siguiente ejemplo.

```
<form name="formulario" method="get" action="procesar.php">
  <p><label>Nombre: <input type="text" name="nombre"
maxlength="15"></label></p>
  <p><label><input type="radio" name="edad" value="15" checked> 15
Años</label></p>
  <p><label><input type="radio" name="edad" value="30"> 30
Años</label></p>
  <p><label><input type="radio" name="edad" value="45"> 45
Años</label></p>
  <p><label><input type="radio" name="edad" value="60"> 60
Años</label></p>
  <p><input type="submit" value="Enviar"></p>
</form>
```

Listado 2-55: *Implementando el tipo* `radio`

En el formulario del Listado 2-55 declaramos cuatro elementos **`<input>`** de tipo **`radio`** para ofrecer distintas edades que el usuario puede elegir. Como todos los elementos tienen el mismo nombre (**`edad`**), se consideran parte del mismo grupo y, por lo tanto, solo una de las opciones

puede ser seleccionada a la vez. Además del atributo **name**, también implementamos un atributo booleano llamado **checked**. Este atributo le dice al navegador que seleccione el botón cuando se carga el documento, lo cual determina la edad de 15 años como el valor por defecto.

Figura 2-33: *Campos de entrada de tipo* `radio`

Como ya mencionamos, solo se puede seleccionar uno de estos botones a la vez. El valor asignado al atributo **value** del elemento seleccionado es el que se enviará al servidor. El tipo **checkbox** genera un tipo de entrada similar. En este caso, el usuario puede seleccionar múltiples valores haciendo clic en las casillas correspondientes.

```
<form name="formulario" method="get" action="procesar.php">
  <p><label>Nombre: <input type="text" name="nombre"
maxlength="15"></label></p>
  <p><label><input type="checkbox" name="edad15" value="15" checked> 15
Años</label></p>
  <p><label><input type="checkbox" name="edad30" value="30" checked> 30
Años</label></p>
  <p><label><input type="checkbox" name="edad45" value="45"> 45
Años</label></p>
  <p><label><input type="checkbox" name="edad60" value="60"> 60
Años</label></p>
  <p><input type="submit" value="Enviar"></p>
</form>
```

Listado 2-56: *Implementando el tipo* `checkbox`

El tipo **checkbox** es similar al tipo **radio**, pero tenemos que asignar diferentes nombres a cada elemento porque el usuario puede seleccionar varias opciones al mismo tiempo. Cuando el usuario selecciona una o más opciones, los valores de todos esos elementos se envían al servidor. Esta clase de campo de entrada también puede incluir el atributo **checked** para seleccionar opciones por defecto. En nuestro ejemplo seleccionamos dos valores: 15 y 30.

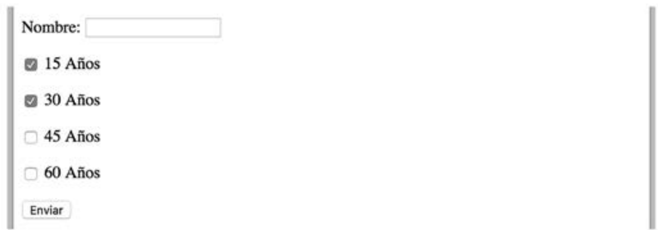

Figura 2-34: *Campos de entrada de tipo* `checkbox`

Otro tipo de entrada que genera una herramienta visual es **date**. Este control le ayuda al usuario a seleccionar una fecha. Algunos navegadores lo implementan como un calendario que se muestra cada vez que el usuario hace clic en el campo. El valor enviado al servidor por este tipo de campos tiene la sintaxis año-mes-día, por lo que si queremos especificar un valor inicial o el navegador no facilita una herramienta para seleccionarlo, debemos declararlo en este formato.

```
<input type="date" name="fecha" value="2017-02-05">
```

Listado 2-57: *Implementando el tipo* date

El elemento **<input>** en el Listado 2-57 crea un campo de entrada con la fecha 2017-02-05 como valor por defecto. Si este elemento se muestra en un navegador que facilita un calendario para seleccionar la fecha, como Google Chrome, la fecha inicial seleccionada en el calendario será la que defina el atributo **value**.

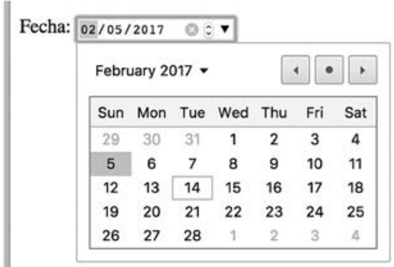

Figura 2-35: *Campo de entrada de tipo* date

El tipo **date** no es el único disponible para insertar fechas. HTML también ofrece el tipo **datetime-local** para seleccionar fecha y hora, el tipo **week** para seleccionar una semana, el tipo **month** para seleccionar un mes, y el tipo **time** para seleccionar horas y minutos. Estos tipos se crearon con diferentes propósitos y, por lo tanto, esperan valores con diferentes sintaxis. El tipo **datetime-local** espera un valor que representa la fecha y la hora en el formato año-mes-día horas:minutos:segundos, con la letra T separando la fecha de la hora, como en 2017-02-05T10:35:00. El tipo **week** espera un valor con la sintaxis **2017-W30**, donde **2017** es el año y **30** es el número de la semana. El tipo **month** espera una sintaxis año-mes. Y finalmente, el tipo **time** espera un valor que representa horas y minutos con la sintaxis horas:minutos (el carácter : es convertido a la cadena de caracteres %3ª cuando el valor se envía al servidor).

Hágalo usted mismo: reemplace los elementos **<input>** en su archivo HTML por el elemento del Listado 2-57 y abra el documento en su navegador. Haga clic en el elemento. En un navegador moderno, debería ver una ventana con un calendario donde puede seleccionar una fecha. Cambie el tipo de entrada en el elemento **<input>** por cualquiera de los tipos mencionados arriba para ver las clases de herramientas que facilita el navegador para introducir estos tipos de valores. Si pulsa el botón **Enviar**, el valor seleccionado o introducido en el campo se agrega a la URL y el navegador intenta acceder al archivo procesar.php en el servidor para procesarlo. Debido a que este archivo aún no existe, el navegador devuelve

un error, pero al menos podrá ver en la barra de navegación cómo se incluyen los valores en la URL. Más adelante en este capítulo estudiaremos el modo de enviar un formulario y cómo procesar los valores en el servidor.

Además de los tipos de campos de entrada disponibles para introducir fechas, existe un tipo de campo llamado **color** que ofrece una interfaz predefinida para seleccionar un valor. Generalmente, el valor esperado por estos campos es un número hexadecimal, como #00FF00.

```
<input type="color" name="micolor" value="#99BB00">
```

Listado 2-58: *Implementando el tipo* color

Un elemento **<input>** de tipo **color** se presenta en la pantalla como un rectángulo pintado del color determinado por defecto, seleccionado por el usuario, o definido por el atributo **value**. Cuando el usuario hace clic en este rectángulo, el navegador abre una herramienta que nos permite seleccionar un nuevo color.

Figura 2-36: *Campo de entrada de tipo* color

Lo básico: los colores se pueden expresar en varios formatos y seleccionar desde sistemas de colores diferentes. La nomenclatura más común es la hexadecimal, cuya sintaxis incluye el carácter # seguido por tres valores hexadecimales que representan los componentes del color (rojo, verde, y azul). Estudiaremos cómo definir colores en el Capítulo 3.

Hasta el momento hemos usado un elemento **<input>** de tipo **submit** para crear el botón que el usuario tiene que pulsar para enviar el formulario. Este tipo de entrada crea un botón estándar identificado con un título. Si queremos mejorar el diseño, podemos implementar el tipo de entrada **image** que crea un botón con una imagen. Estos tipos de campos requieren el atributo **src** con la URL del archivo que contiene la imagen que queremos usar para el botón, y pueden incluir los atributos **width** y **height** para definir su tamaño.

```
<form name="formulario" method="get" action="procesar.php">
  <p><label>Name: <input type="text" name="nombre"></label></p>
  <p><label>Age: <input type="text" name="edad"></label></p>
  <p><input type="image" src="botonenviar.png" width="100"></p>
</form>
```

Listado 2-59: *Implementando el tipo* image *para crear un botón personalizado*

El formulario del Listado 2-59 crea el botón para enviar el formulario con un elemento **<input>** de tipo **image**. El elemento carga la imagen del archivo botonenviar.png y la

muestra en la pantalla. Cuando el usuario hace clic en la imagen, el formulario se envía del mismo modo que con los botones que hemos usado anteriormente.

Figura 2-37: *Campo de entrada de tipo* `image`

Hágalo usted mismo: reemplace el formulario en su archivo HTML por el formulario del Listado 2-59. Descargue la imagen botonenviar.png desde nuestro sitio web. Abra el documento en su navegador. La imagen tiene un tamaño de 150 píxeles, pero como declaramos el atributo **`width`** con un valor de 100, el navegador reduce el ancho de la imagen a 100 píxeles. Elimine este atributo para ver la imagen en sus dimensiones originales.

Aunque los botones creados con elementos **`<input>`** son probablemente más que suficientes para la mayoría de los proyectos, HTML ofrece un elemento más versátil para crear botones llamado **`<button>`**. Este elemento incluye el atributo **`type`** para determinar el tipo de botón que queremos generar. Por ejemplo, si queremos crear un botón para enviar el formulario, debemos declarar el valor **`submit`**.

```
<form name="formulario" method="get" action="procesar.php">
  <p><label>Nombre: <input type="text" name="nombre"></label></p>
  <p><label>Edad: <input type="text" name="edad"></label></p>
  <p><button type="submit">Enviar Formulario</button></p>
</form>
```

Listado 2-60: *Implementando el elemento* `<button>` *para crear un botón*

Lo básico: el elemento **`<button>`** crea un botón estándar con las mismas características que el botón creado por el elemento **`<input>`**. La diferencia es que el título de estos botones se define entre las etiquetas de apertura y cierre, lo cual nos permite usar otros elementos HTML e incluso imágenes para declararlo. Por esta razón, el elemento **`<button>`** es el que se prefiere cuando queremos personalizarlo usando estilos CSS o cuando queremos usarlo para ejecutar códigos JavaScript, como veremos en próximos capítulos.

El elemento **`<input>`** permite al usuario insertar o seleccionar varios tipos de valores, pero los campos de entrada generados por este elemento nos dejan introducir una sola línea de texto. HTML ofrece el elemento **`<textarea>`** para insertar múltiples líneas de texto. El elemento está compuesto por etiquetas de apertura y cierre, y puede incluir los atributos **`rows`** y **`cols`** para definir el ancho y la altura del área en caracteres.

```
<form name="formulario" method="get" action="procesar.php">
  <p><label>Texto: <textarea name="texto" cols="50"
rows="6"></textarea></label></p>
  <p><input type="submit" value="Enviar"></p>
</form>
```

Listado 2-61: *Implementando el elemento* `<textarea>`

En la ventana del navegador, el elemento **<textarea>** se representa por un recuadro vacío del tamaño determinado por sus atributos o los estilos por defecto. El texto insertado queda limitado por el ancho del área, pero se puede extender verticalmente todo lo que sea necesario. Si el área no es lo suficientemente larga para contener el texto completo, el navegador muestra barras laterales con las que el usuario puede desplazar el contenido.

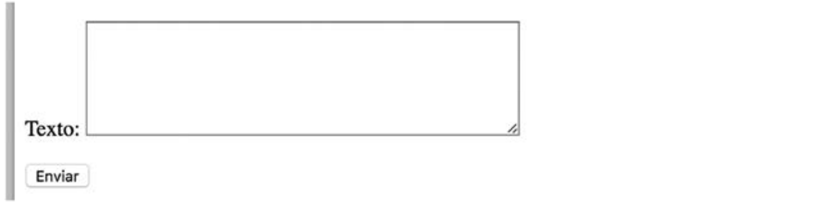

Figura 2-38: *El elemento* `<textarea>`

Hágalo usted mismo: reemplace el formulario en su archivo HTML por el formulario del Listado 2-61 y abra el documento en su navegador. Escriba varias líneas de texto para ver cómo se distribuye el texto dentro del área.

Lo básico: si necesita declarar un valor inicial para el elemento **<textarea>**, puede insertarlo entre las etiquetas de apertura y cierre.

Además de los tipos de campos de entrada **radio** y **checkbox** estudiados anteriormente, HTML ofrece el elemento **<select>** para presentar una lista de valores al usuario. Cuando el usuario hace clic en este elemento, la lista se muestra en una ventana desplegable, y luego el valor seleccionado por el usuario se inserta en el campo. Debido a que el elemento **<select>** no genera un campo de entrada, el usuario no puede insertar valores distintos de los incluidos en la lista.

El elemento **<select>** trabaja junto con el elemento **<option>** para definir las opciones. El elemento **<select>** debe incluir el atributo **name** para identificar el valor, y cada elemento **<option>** debe incluir el atributo **value** para definir el valor que representa.

```
<form name="formulario" method="get" action="procesar.php">
  <p>
    <label for="listado">Libros: </label>
    <select name="libro" id="listado">
      <option value="1">IT</option>
      <option value="2">Carrie</option>
      <option value="3">El Resplandor</option>
      <option value="4">Misery</option>
    </select>
  </p>
```

```
  <p><input type="submit" value="Enviar"></p>
</form>
```

Listado 2-62: *Implementando el elemento* `<select>`

En el ejemplo del Listado 2-62, incluimos cuatro opciones, una para cada libro que queremos mostrar en la lista. Los valores de las opciones son definidos desde 1 a 4. Por consiguiente, cada vez que el usuario selecciona un libro y envía el formulario, el número correspondiente a ese libro se envía al servidor con el identificador "libro" (el valor del atributo **`name`**).

Figura 2-39: *Seleccionando un valor con el elemento* `<select>`

Otra manera de crear una lista predefinida es con el elemento **`<datalist>`**. Este elemento define una lista de ítems que, con la ayuda del atributo **`list`**, se puede usar como sugerencia en un campo de entrada. Al igual que las opciones para el elemento **`<select>`**, las opciones para este elemento se definen por elementos **`<option>`**, pero en este caso deben incluir el atributo **`label`** con una descripción del valor. El siguiente ejemplo crea un formulario con un campo de entrada para insertar un número telefónico. El código incluye un elemento **`<datalist>`** con dos elementos **`<option>`** que definen los valores que queremos sugerir al usuario.

```
<form name="formulario" method="get" action="procesar.php">
  <datalist id="datos">
    <option value="123123123" label="Teléfono 1">
    <option value="456456456" label="Teléfono 2">
  </datalist>
  <p><label>Teléfono: <input type="tel" name="telefono"
list="datos"></label></p>
  <p><input type="submit" value="Enviar"></p>
</form>
```

Listado 2-63: *Sugiriendo una lista de valores con el elemento* `<datalist>`

Para conectar un campo de entrada con un elemento **`<datalist>`**, tenemos que incluir el atributo **`list`** en el elemento **`<input>`** con el mismo valor que usamos para identificar el elemento **`<datalist>`**. Para este propósito, en el formulario del Listado 2-63 incluimos el atributo **`id`** en el elemento **`<datalist>`** con el valor "datos" y luego asignamos este mismo valor al atributo **`list`** del elemento **`<input>`**. En consecuencia, el campo de entrada muestra una flecha que despliega una lista de valores predefinidos que el usuario puede seleccionar para completar el formulario.

Figura 2-40: *Valores predefinidos con el elemento* `<datalist>`

Hágalo usted mismo: reemplace el formulario en su archivo HTML con el formulario del Listado 2-63. Abra el documento en su navegador y haga clic en la flecha del lado derecho del campo. Debería ver algo similar a lo que se muestra en la Figura 2-40.

HTML incluye otros dos elementos que podemos usar en un formulario: **<progress>** y **<meter>**, que no se consideran elementos de formulario, pero debido a que representan medidas, son realmente útiles cuando nuestro formulario produce esta clase de información.

El elemento **<progress>** se usa para informar del progreso en la ejecución de una tarea. Requiere dos atributos que determinan el valor actual y el máximo. El atributo **value** indica el progreso logrado hasta el momento, y el atributo **max** declara el valor que necesitamos alcanzar para dar por finalizada la tarea.

```
<progress value="30" max="100">0%</progress>
```

Listado 2-64: *Implementando el elemento* <progress>

Los navegadores representan este elemento con una barra de dos colores. Por defecto, la porción de la barra que representa el progreso se muestra en color azul. Si el navegador no reconoce el elemento, el valor entre las etiquetas de apertura y cierre se muestra en su lugar.

Figura 2-41: *Barra de progreso*

Al igual que el elemento **<progress>**, el elemento **<meter>** se usa para mostrar una escala, pero no representa progreso. Su propósito es representar un rango de valores predefinidos (por ejemplo, el espacio ocupado en un disco duro). Este elemento cuenta con varios atributos asociados. Los atributos **min** y **max** determinan los límites del rango, **value** determina el valor medido, y **low**, **high** y **optimum** se usan para segmentar el rango en secciones diferenciadas y declarar la posición óptima.

```
<meter value="60" min="0" max="100" low="40" high="80"
optimum="100">60</meter>
```

Listado 2-65: *Implementando el elemento* <meter>

El código del Listado 2-65 genera una barra en la pantalla que muestra un nivel de 60 en una escala de 0 a 100 (de acuerdo con los valores declarados por los atributos **value**, **min** y **max**). El color de la barra generada por el elemento depende de los niveles determinados por los atributos **low**, **high** y **optimum**. Debido a que el valor actual en nuestro ejemplo se encuentra entre los valores de los atributos **low** y **high**, la barra se muestra en color amarillo.

Figura 2-42: *Barra creada con el elemento* <meter>

Enviando el formulario

Al enviar un formulario, los datos introducidos se envían al servidor usando la sintaxis nombre/valor para cada elemento, donde *nombre* es el valor asignado al atributo **name** del elemento y *valor* es el valor que introduce el usuario. Por ejemplo, si insertamos el texto "Roberto" en un campo de entrada con el nombre "minombre", el par nombre/valor que se envía al servidor será minombre/Roberto.

Los navegadores usan dos métodos para enviar esta información: GET y POST. El método se declara asignando los valores **GET** o **POST** al atributo **method** del elemento **<form>**, como hemos hecho en ejemplos anteriores, aunque el método a usar depende del tipo de información gestionada por el formulario. Como mencionamos en el Capítulo 1, los navegadores se comunican con el servidor usando un protocolo llamado HTTP. Cuando el navegador quiere acceder a un documento, envía una solicitud HTTP al servidor. Una solicitud HTTP es un mensaje que le dice al servidor cuál es el recurso al que el navegador quiere acceder y lo que quiere hacer con el mismo (descargar el documento, procesar información, etc.). El mensaje está compuesto por la URL del recurso, los datos asociados con la solicitud, como la fecha y el idioma, y un cuerpo con información adicional. Los pares nombre/valor producidos por un formulario se envían al servidor dentro de estas solicitudes HTTP. Si el método se ha declarado como **GET**, los pares nombre/valor se agregan al final de la URL, pero si el método se ha declarado como **POST**, los valores se incluyen en el cuerpo de la solicitud. Esto significa que la información enviada con el método GET es visible para el usuario (el usuario puede ver la URL con todos los pares nombre/valor en la barra de navegación del navegador), pero la información enviada con el método POST se oculta dentro de la solicitud. En consecuencia, si la información es sensible o privada, debemos usar el método POST, pero si la información no es sensible, como valores de búsqueda insertados por el usuario, podemos usar el método GET.

Así mismo, tenemos que considerar que el método POST se puede usar para enviar una cantidad ilimitada de información, pero el método GET tiene que adaptarse a las limitaciones presentadas por las URL. Esto se debe a que el largo de una URL es limitado. Si la información insertada en el formulario es muy extensa, se podría perder.

El siguiente ejemplo presenta un formulario con un único campo de entrada para ilustrar cómo funciona este proceso. El elemento **<input>** se identifica con el nombre "val", y el método utilizado para enviar la información se declara como GET, lo que significa que el valor insertado por el usuario se agregará a la URL.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p><input type="text" name="val"></p>
      <p><input type="submit" value="Enviar"></p>
    </form>
  </section>
</body>
</html>
```

Listado 2-66: *Formularios con el método GET*

En el ejemplo del Listado 2-66, un archivo llamado procesar.php se declara a cargo de procesar la información. Cuando el usuario pulsa el botón **Enviar**, el navegador crea una solicitud HTTP que incluye la URL que apunta a este archivo. Debido a que el método del formulario se ha declarado como GET, el nombre del campo de entrada y el valor insertado por el usuario se agregan a la URL, tal como se muestra a continuación.

Figura 2-43: *Par nombre/valor en la URL*

Cuando la información se envía con el método GET, los pares nombre/valor se agregan al final de la URL separados por el carácter =, y el primer par es precedido por el carácter ?. Si existe más de un par de valores, los restantes se agregan a la URL separados por el carácter &, como en www.ejemplo.com/procesar.php?val1=10&val2=20.

Al otro lado, el servidor recibe esta solicitud, lee la URL, extrae los valores y ejecuta el código en el archivo procesar.php. Este código debe procesar la información recibida y producir una respuesta. La forma en la que se realiza esta tarea depende del lenguaje de programación que utilizamos y lo que queremos lograr. Por ejemplo, para leer los valores enviados con el método GET, PHP los ofrece en un listado llamado **`$_GET`**. La sintaxis requerida para obtener el valor incluye su nombre entre corchetes.

```
<?php
  print('El valor es: '.$_GET['val']);
?>
```

Listado 2-67: *Procesando los datos en el servidor con PHP (procesar.php)*

El ejemplo del Listado 2-67 muestra cómo procesar valores con PHP. Las etiquetas **`<?php`** y **`?>`** indican al servidor que este es código PHP y que tiene que ser ejecutado como tal. El código puede ser extenso o estar compuesto por solo unas pocas instrucciones, dependiendo de lo que necesitamos. Nuestro ejemplo incluye una sola instrucción para ilustrar cómo se procesa la información. Esta instrucción, llamada **`print()`**, toma los valores entre paréntesis y los incluye en un archivo que se va a devolver al navegador como respuesta. En este caso, agregamos el valor recibido por medio de la solicitud al texto "El valor es: ". Si enviamos el valor 10, como en el ejemplo de la Figura 2-43, el servidor genera un archivo con el texto "El valor es: 10" y lo envía de regreso al navegador.

Lo básico: el código PHP de nuestro ejemplo utiliza el nombre **`$_GET`** para capturar la información recibida desde el navegador porque el método del formulario se ha declarado como GET, pero si cambiamos el método a POST, debemos utilizar el nombre **`$_POST`**.

El archivo producido por el servidor a través de un código PHP es un archivo HTML. Por propósitos didácticos, no incluimos ningún elemento HTML en el ejemplo del Listado 2-67, pero siempre deberíamos generar un documento HTML válido en respuesta. Existen varias maneras de definir un documento en PHP. La más simple es crear el documento HTML como lo hemos hecho anteriormente pero dentro de un archivo PHP, e incluir el código PHP donde queremos mostrar el resultado. Por ejemplo, el siguiente ejemplo inserta código PHP dentro de un elemento **`<p>`**.

Cuando el servidor abre el archivo procesar.php con este documento, ejecuta el código PHP, inserta el resultado dentro de las etiquetas **<p>**, y devuelve el archivo al servidor. El resultado es el mismo que si hubiéramos creado un documento estático con el texto "El valor es: 10", pero el texto se genera dinámicamente en el servidor con los valores recibidos desde el formulario.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Respuesta</title>
</head>
<body>
  <section>
    <p>
      <?php
        print('El valor es: '.$_GET['val']);
      ?>
    </p>
  </section>
</body>
</html>
```

Listado 2-68: *Generando un documento HTML que incluye código PHP (procesar.php)*

Hágalo usted mismo: para probar este ejemplo, tiene que subir los archivos a un servidor que pueda ejecutar código PHP. Si no posee una cuenta de alojamiento con estas características, puede instalar un servidor en su ordenador con paquetes como MAMP, introducido en el Capítulo 1 (si no sabe cómo usar un servidor, no se preocupe, este procedimiento no es necesario para realizar la mayoría de los ejemplos de este libro). Cree un nuevo archivo HTML con el código del Listado 2-66, y un archivo llamado procesar.php con el código del Listado 2-67 o 2-68. Suba los archivos al servidor y abra el documento en su navegador. Inserte un valor dentro del campo de entrada y pulse el botón Enviar. El navegador debería mostrar una nueva página que contenga el texto que comienza con la cadena de caracteres "El valor es: " y termina con el valor que ha insertado en el formulario.

IMPORTANTE: este es simplemente un ejemplo de cómo se procesa la información y cómo el navegador transmite los datos introducidos en el formulario al servidor. Estudiaremos otros ejemplos parecidos en próximos capítulos, pero no es el propósito de este libro enseñar cómo programar en PHP u otro lenguaje de programación de servidor. Para aprender más sobre PHP, vaya a nuestro sitio web y visite las secciones Enlaces y Vídeos.

Atributos globales

HTML define atributos globales que son exclusivos de elementos de formulario. Los siguientes son los más utilizados.

disabled—Este es un atributo booleano que desactiva el elemento. Cuando el atributo está presente, el usuario no puede introducir valores o interactuar con el elemento.

readonly—Este atributo indica que el valor del elemento no se puede modificar.

placeholder—Este atributo muestra un texto en el fondo del elemento que indica al usuario el valor que debe introducir.

autocomplete—Este atributo activa o desactiva la función de autocompletar. Los valores disponibles son `on` y `off`.

novalidate—Este es un atributo booleano para el elemento `<form>` que indica que el formulario no debería ser validado.

formnovalidate—Este es un atributo booleano para los elementos `<button>` e `<input>` de tipo `submit` e `image` que indica que el formulario al que pertenecen no debería ser validado.

required—Este es un atributo booleano que indica al navegador que el usuario debe seleccionar o insertar un valor en el elemento para validar el formulario.

multiple—Este es un atributo booleano que indica al navegador que se pueden insertar múltiples valores en el campo (se aplica a elementos `<input>` de tipo `email` y `file`).

autofocus—Este atributo booleano solicita al navegador que mueva el foco al elemento tan pronto como se carga el documento.

pattern—Este atributo define una expresión regular que el navegador debe usar para validar el valor insertado en el campo.

form—Este atributo asocia el elemento con un formulario. Se usa para conectar un elemento con un formulario cuando el elemento no se define entre las etiquetas `<form>`. El valor asignado a este atributo debe ser el mismo asignado al atributo `id` del elemento `<form>`.

spellcheck—Este atributo solicita al navegador que compruebe la ortografía y gramática del valor introducido en el campo. Los valores disponibles son `true` (verdadero) y `false` (falso).

Los atributos `disabled` y `readonly` tienen un propósito similar, no permitir al usuario interactuar con el elemento, pero se aplican en diferentes circunstancias. El atributo `disabled` normalmente se implementa cuando queremos mostrar al usuario que el elemento puede estar disponible en otras condiciones, como cuando el control no es aplicable en el país del usuario, por ejemplo. Por otro lado, el atributo `readonly` se implementa cuando solo existe un valor posible y no queremos que el usuario lo cambie. Por ejemplo, en el siguiente formulario, el usuario no puede introducir la edad.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p><label>Nombre: <input type="text" name="nombre"></label></p>
```

```
      <p><label>Edad: <input type="text" name="edad" disabled></label></p>
      <p><input type="submit" value="Enviar"></p>
    </form>
  </section>
</body>
</html>
```

Listado 2-69: *Implementando el atributo* `disabled`

Los elementos afectados por los atributos **`disabled`** y **`readonly`** se muestran en colores más claros para advertir al usuario de que no son controles normales. Otro atributo que afecta la apariencia de un elemento es **`placeholder`**. Este se usa en campos de entrada para ofrecer una pista (una palabra o frase) que ayude al usuario a introducir el valor correcto. El siguiente ejemplo inserta esta ayuda en un campo de búsqueda.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p><label>Buscar: <input type="search" name="buscar"
placeholder="Término a buscar"></label></p>
      <p><input type="submit" value="Buscar"></p>
    </form>
  </section>
</body>
</html>
```

Listado 2-70: *Implementando el atributo* `placeholder`

El valor de este atributo lo muestran los navegadores dentro del campo hasta que el usuario inserta un valor.

Figura 2-44: *Campo de entrada con un mensaje de ayuda*

Una de las características de los formularios es que tienen la capacidad de validar los datos introducidos. Por defecto, los formularios validan los datos a menos que el atributo **`novalidate`** sea declarado. Este atributo booleano es específico de elementos **`<form>`**. Cuando es incluido, el formulario se envía sin validar. La presencia de este atributo afecta al formulario de forma permanente, pero a veces el proceso de validación es requerido solo en ciertas circunstancias. Por ejemplo, cuando la información insertada debe ser grabada para permitir al usuario continuar con el trabajo más adelante. En casos como este, podemos

implementar el atributo **formnovalidate**. Este atributo está disponible para los elementos que crean los botones para enviar el formulario. Cuando los datos se envían con un botón que contiene este atributo, el formulario no es validado.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p><label>Correo: <input type="email" name="correo"></label></p>
      <p>
        <input type="submit" value="Enviar">
        <input type="submit" value="Grabar" formnovalidate>
      </p>
    </form>
  </section>
</body>
</html>
```

Listado 2-71: *Enviando un formulario sin validar con el atributo* `formnovalidate`

En el ejemplo del Listado 2-71, el formulario será validado en circunstancias normales, pero incluimos un segundo botón con el atributo **formnovalidate** para poder enviar el formulario sin pasar por el proceso de validación. El botón **Enviar** requiere que el usuario introduzca una cuenta de correo válida, pero el botón **Grabar** no incluye este requisito.

Cuando usamos el tipo **email** para recibir una cuenta de correo, como en el ejemplo anterior, el navegador controla si el valor introducido es una cuenta de correo, pero valida el campo cuando se encuentra vacío. Esto se debe a que el campo no es obligatorio. HTML ofrece el atributo **required** para cambiar esta condición. Cuando se incluye el atributo **required**, el campo solo será válido si el usuario introduce un valor y este valor cumple con los requisitos de su tipo. El siguiente ejemplo implementa este atributo para forzar al usuario a introducir un correo electrónico.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p><label>Correo: <input type="email" name="correo"
required></label></p>
      <p>
        <input type="submit" value="Enviar">
        <input type="submit" value="Grabar" formnovalidate>
      </p>
```

```
    </form>
  </section>
</body>
</html>
```

Listado 2-72: *Declarando una entrada* `email` *como campo requerido*

Hágalo usted mismo: cree un archivo HTML con el código del Listado 2-72. Abra el documento en su navegador y pulse el botón enviar. Debería recibir un mensaje de error en el campo **`correo`** indicando que debe insertar un valor. Pulse el botón **Grabar**. Como este botón incluye el atributo **`formnovalidate`**, el error ya no se muestra y se envía el formulario.

Otro atributo que se usa para validación es **`pattern`**. Algunos tipos de campos de entrada validan cadenas de caracteres específicas, pero no pueden hacer nada cuando el valor no es estándar, como en el caso de los códigos postales. No existe un tipo de campo predeterminado para esta clase de valores. El atributo **`pattern`** nos permite crear un filtro personalizado usando expresiones regulares.

Las expresiones regulares son textos compuestos por una serie de caracteres que definen un patrón de concordancia. Por ejemplo, los caracteres 0-9 determinan que solo se aceptan los números entre 0 y 9. Utilizando esta clase de expresiones, podemos crear un filtro personalizado para validar cualquier valor que necesitemos. El siguiente formulario incluye un campo de entrada que solo acepta números con cinco dígitos.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p><label>Código Postal: <input pattern="[0-9]{5}" name="cp"
title="Inserte su código postal"></label></p>
      <p><input type="submit" value="Enviar"></p>
    </form>
  </section>
</body>
</html>
```

Listado 2-73: *Personalizando campos de entrada con el atributo* `pattern`

El elemento **`<input>`** en el Listado 2-73 incluye el atributo **`pattern`** con el valor **`[0-9]{5}`**. Esta expresión regular determina que el valor debe tener exactamente cinco caracteres y que esos caracteres deben ser números entre 0 y 9. En consecuencia, solo podemos insertar números de cinco dígitos; cualquier otro carácter o tamaño devolverá un error.

Estos tipos de entrada también pueden incluir el atributo **`title`** para explicar al usuario cuál es el valor esperado. Este mensaje complementa el mensaje de error estándar que muestra el navegador, tal como ilustra la Figura 2-45.

Figura 2-45: *Error en un campo de entrada con un patrón personalizado*

IMPORTANTE: no es el propósito de este libro enseñar cómo trabajar con expresiones regulares. Para más información, visite nuestro sitio web y siga los enlaces de este capítulo.

Los tipos de entrada **`email`** y **`file`** solo permiten al usuario introducir un valor a la vez. Si queremos permitir la inserción de múltiples valores, tenemos que incluir el atributo **`multiple`**. Con este atributo, el usuario puede insertar todos los valores que quiera separados por coma. El siguiente ejemplo incluye un campo de entrada que acepta múltiples direcciones de correo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p><label>Correo: <input type="email" name="correo"
multiple></label></p>
      <p><input type="submit" value="Enviar"></p>
    </form>
  </section>
</body>
</html>
```

Listado 2-74: *Declarando un campo de entrada* `email` *como campo múltiple*

Además de los atributos de validación, existen otros atributos que pueden ayudar al usuario a decidir qué valores introducir. Por ejemplo, el atributo **`autocomplete`** activa una herramienta que le sugiere al usuario qué introducir según los valores insertados previamente. Los valores disponibles para este atributo son **`on`** y **`off`** (activar y desactivar la herramienta). Este atributo también se puede implementar en el elemento **`<form>`** para que afecte a todos los elementos del formulario. El siguiente formulario desactiva las sugerencias para una búsqueda.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
```

```
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p><label>Buscar: <input type="search" name="buscar"
autocomplete="off"></label></p>
      <p><input type="submit" value="Buscar"></p>
    </form>
  </section>
</body>
</html>
```

Listado 2-75: *Implementando el atributo* `autocomplete`

Hágalo usted mismo: cree un archivo HTML con el código del Listado 2-75. Abra el documento en su navegador. Inserte un valor y presione el botón **Enviar**. Repita el proceso. El navegador no debería sugerir ningún valor a insertar. Cambie el valor del atributo **`autocomplete`** a **`on`** y repita el proceso. En esta oportunidad, cada vez que introduzca un valor, el navegador le mostrará un listado con los valores insertados previamente que comienzan con los mismos caracteres.

Otro atributo que puede ayudar al usuario a decidir qué insertar es **`autofocus`**. En este caso, el atributo establece el foco en el elemento cuando se carga el documento, sugiriendo al usuario qué valor insertar primero. El siguiente ejemplo incluye dos campos de entrada para insertar el nombre y la edad del usuario, pero el campo para la edad incluye el atributo **`autofocus`** y, por lo tanto, el navegador posicionará el cursor en este campo por defecto.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p><label>Nombre: <input type="text" name="nombre"></label></p>
      <p><label>Edad: <input type="text" name="edad" autofocus></label></p>
      <p><input type="submit" value="Enviar"></p>
    </form>
  </section>
</body>
</html>
```

Listado 2-76: *Implementando el atributo* `autofocus`

Otra herramienta que pueden activar automáticamente los navegadores es el control de ortografía. A pesar de la utilidad de esta herramienta y de que los usuarios esperan casi siempre tenerla a su disposición, puede resultar inapropiada en ciertas circunstancias. La herramienta se activa por defecto, pero podemos incluir el atributo **`spellcheck`** con el valor **`false`** para desactivarla, como hacemos en el siguiente ejemplo.

```
<!DOCTYPE html>
<html lang="es">
```

```
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p><label>Texto: <textarea name="texto" cols="50" rows="6"
spellcheck="false"></textarea></label></p>
      <p><input type="submit" value="Enviar"></p>
    </form>
  </section>
</body>
</html>
```

Listado 2-77: *Desactivando el control de ortografía*

Finalmente, existe otro atributo útil que podemos implementar para declarar como parte del formulario elementos que no se han incluido entre las etiquetas **<form>**. Por ejemplo, si tenemos un campo de entrada en el área de navegación, pero queremos que el elemento se envíe con el formulario definido en el área central del documento, podemos conectar este elemento con el formulario usando el atributo **form** (el valor de este atributo debe coincidir con el valor del atributo **id** asignado al elemento **<form>**).

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
</head>
<body>
  <nav>
    <p><input type="search" name="buscar" form="formulario"></p>
  </nav>
  <section>
    <form name="formulario" id="formulario" method="get"
action="procesar.php">
      <p><input type="submit" value="Enviar"></p>
    </form>
  </section>
</body>
</html>
```

Listado 2-78: *Declarando elementos de formulario en una parte diferente del documento*

IMPORTANTE: los formularios son extremadamente útiles en el desarrollo web, y se requieren en la mayoría de los sitios y aplicaciones web modernas. Los ejemplos de este capítulo se han creado con propósitos didácticos y se han simplificado al máximo para que pueda probar los elementos y controles disponibles. En próximos capítulos, estudiaremos cómo implementar formularios en situaciones más prácticas y expandirlos con CSS y JavaScript.

Capítulo 3

CSS

3.1 Estilos

En el capítulo anterior, hemos aprendido a crear un documento HTML, a organizar su estructura, y a determinar qué elementos son más apropiados para representar su contenido. Esta información nos permite definir el documento, pero no determina cómo se mostrará en pantalla. Desde la introducción de HTML5, esa tarea es responsabilidad de CSS. CSS es un lenguaje que facilita instrucciones que podemos usar para asignar estilos a los elementos HTML, como colores, tipos de letra, tamaños, etc. Los estilos se deben definir con CSS y luego asignar a los elementos hasta que logramos el diseño visual que queremos para nuestra página.

Por razones de compatibilidad, los navegadores asignan estilos por defecto a algunos elementos HTML. Esta es la razón por la que en el Capítulo 2 algunos elementos tenían márgenes o generaban saltos de línea, pero otros eran definidos de forma parecida (tenían el mismo tipo de letra y colores, por ejemplo). Algunos de estos estilos son útiles, pero la mayoría deben ser reemplazados o complementados con estilos personalizados. En CSS, los estilos personalizados se declaran con propiedades. Un estilo se define declarando el nombre de la propiedad y su valor separados por dos puntos. Por ejemplo, el siguiente código declara una propiedad que cambia el tamaño de la letra a 24 píxeles (debido a que algunas propiedades pueden incluir múltiples valores separados por un espacio, debemos indicar el final de la línea con un punto y coma).

```
font-size: 24px;
```

Listado 3-1: *Declarando propiedades CSS*

Si la propiedad del Listado 3-1 se aplica a un elemento, el texto contenido por ese elemento se mostrará en la pantalla con el tipo de letra definido por defecto, pero con un tamaño de 24 píxeles.

Lo básico: aunque los píxeles (`px`) son las unidades de medida que más se implementan, más adelante veremos que en algunas circunstancias, especialmente cuando creamos nuestro sitio web con diseño web adaptable, pueden ser más apropiadas otras unidades. Las unidades más utilizadas son `px` (píxeles), `pt` (puntos), `in` (pulgadas), `cm` (centímetros), `mm` (milímetros), `em` (relativo al tamaño de letra del elemento), `rem` (relativo al tamaño de letra del documento), y `%` (relativo al tamaño del contenedor del elemento).

La mayoría de las propiedades CSS pueden modificar un solo aspecto del elemento (el tamaño de la letra en este caso). Si queremos cambiar varios estilos al mismo tiempo, tenemos que declarar múltiples propiedades. CSS define una sintaxis que simplifica el proceso de asignar múltiples propiedades a un elemento. La construcción se llama *regla*. Una regla es una lista de propiedades declaradas entre llaves e identificadas por un selector. El selector indica qué elementos se verán afectados por las propiedades. Por ejemplo, la siguiente regla

se identifica con el selector **p** y, por lo tanto, sus propiedades se aplicarán a todos los elementos **<p>** del documento.

```
p {
  color: #FF0000;
  font-size: 24px;
}
```

Listado 3-2: *Declarando reglas CSS*

La regla del Listado 3-2 incluye dos propiedades con sus respectivos valores agrupadas por llaves (**color** y **font-size**). Si aplicamos esta regla a nuestro documento, el texto dentro de cada elemento **<p>** se mostrará en color rojo y con un tamaño de 24 píxeles.

Aplicando estilos

Las propiedades y las reglas definen los estilos que queremos asignar a uno o más elementos, pero estos estilos no se aplican hasta que los incluimos en el documento. Existen tres técnicas disponibles para este propósito. Podemos usar estilos en línea, estilos incrustados u hojas de estilo. El primero, estilos en línea, utiliza un atributo global llamado **style** para insertar los estilos directamente en el elemento. Este atributo está disponible en cada uno de los elementos HTML y puede recibir una propiedad o una lista de propiedades que se aplicarán al elemento al que pertenece. Si queremos asignar estilos usando esta técnica, todo lo que tenemos que hacer es declarar el atributo **style** en el elemento que queremos modificar y asignarle las propiedades CSS.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
</head>
<body>
  <main>
    <section>
      <p style="font-size: 20px;">Mi Texto</p>
    </section>
  </main>
</body>
</html>
```

Listado 3-3: Estilos en Línea

El documento del Listado 3-3 incluye un elemento **<p>** con el atributo **style** y el valor **font-size: 20px;**. Cuando el navegador lee este atributo, le asigna un tamaño de 20 píxeles al texto dentro del elemento **<p>**.

Hágalo usted mismo: cree un nuevo archivo HTML con el código del Listado 3-3 y abra el documento en su navegador. Debería ver el texto "Mi Texto" con letras en el tamaño definido por la propiedad **font-size**. Intente cambiar el valor de esta propiedad para ver cómo se presentan en pantalla los diferentes tamaños de letra.

Los estilos en línea son una manera práctica de probar estilos y ver cómo modifican los elementos, pero no se recomiendan para proyectos extensos. La razón es que el atributo **style** solo afecta al elemento en el que se ha declarado. Si queremos asignar el mismo estilo a otros elementos, tenemos que repetir el código en cada uno de ellos, lo cual incrementa innecesariamente el tamaño del documento, complicando su actualización y mantenimiento. Por ejemplo, si más adelante decidimos que en lugar de 20 píxeles el tamaño del texto en cada elemento **<p>** deber ser de 24 píxeles, tendríamos que cambiar cada estilo en cada uno de los elementos **<p>** del documento completo y en todos los documentos de nuestro sitio web.

Una alternativa es la de insertar las reglas CSS en la cabecera del documento usando selectores que determinan los elementos que se verán afectados. Para este propósito, HTML incluye el elemento **<style>**.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <style>
    p {
      font-size: 20px;
    }
  </style>
</head>
<body>
  <main>
    <section>
      <p>Mi texto</p>
    </section>
  </main>
</body>
</html>
```

Listado 3-4: Incluyendo estilos en la cabecera del documento

La propiedad declarada entre las etiquetas **<style>** en el documento del Listado 3-4 cumple la misma función que la declarada en el documento del Listado 3-3, pero en este ejemplo no tenemos que escribir el estilo dentro del elemento **<p>** que queremos modificar porque, debido al selector utilizado, todos se ven afectados por esta regla.

Declarar estilos en la cabecera del documento ahorra espacio y hace que el código sea más coherente y fácil de mantener, pero requiere que copiemos las mismas reglas en cada uno de los documentos de nuestro sitio web. Debido a que la mayoría de las páginas compartirán el mismo diseño, varias de estas reglas se duplicarán. La solución es mover los estilos a un archivo CSS y luego usar el elemento **<link>** para cargarlo desde cada uno de los documentos que lo requieran. Este tipo de archivos se denomina *hojas de estilo*, pero no son más que archivos de texto con la lista de reglas CSS que queremos asignar a los elementos del documento.

Como hemos visto en el Capítulo 2, el elemento **<link>** se usa para incorporar recursos externos al documento. Dependiendo del tipo de recurso que queremos cargar, tenemos que declarar diferentes atributos y valores. Para cargar hojas de estilo CSS, solo necesitamos los atributos **rel** y **href**. El atributo **rel** significa relación y especifica la relación entre el documento y el archivo que estamos incorporando, por lo que debemos declararlo con el valor **stylesheet**

para comunicarle al navegador que el recurso es un archivo CSS con los estilos requeridos para presentar la página. Por otro lado, el atributo **href** declara la URL del archivo a cargar.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <main>
    <section>
      <p>Mi texto</p>
    </section>
  </main>
</body>
</html>
```

Listado 3-5: *Aplicando estilos CSS desde un archivo externo*

El documento del Listado 3-5 carga los estilos CSS desde el archivo misestilos.css. En este archivo tenemos que declarar las reglas CSS que queremos aplicar al documento, tal como lo hemos hecho anteriormente dentro del elemento **<style>**. El siguiente es el código que debemos insertar en el archivo misestilos.css para producir el mismo efecto logrado en ejemplos anteriores.

```
p {
  font-size: 20px;
}
```

Listado 3-6: *Definiendo estilos CSS en un archivo externo*

La práctica de incluir estilos CSS en un archivo aparte es ampliamente utilizada por diseñadores y se recomienda para sitos web diseñados con HTML5, no solo porque podemos definir una sola hoja de estilo y luego incluirla en todos los documentos con el elemento **<link>**, sino porque podemos reemplazar todos los estilos a la vez simplemente cargando un archivo diferente, lo cual nos permite probar diferentes diseños y adaptar el sitio web a las pantallas de todos los dispositivos disponibles, tal como veremos en el Capítulo 5.

Hágalo usted mismo: cree un archivo HTML con el código del Listado 3-5 y un archivo llamado misestilos.css con el código del Listado 3-6. Abra el documento en su navegador. Debería ver el texto dentro del elemento **<p>** con un tamaño de 20 píxeles. Cambie el valor de la propiedad **font-size** en el archivo CSS y actualice la página para ver cómo se aplica el estilo al documento.

Hojas de estilo en cascada

Una característica importante del CSS es que los estilos se asignan en cascada (de ahí el nombre hojas de estilo en cascada o Cascading Style Sheets en inglés). Los elementos en los niveles bajos de la jerarquía heredan los estilos asignados a los elementos en los niveles más

altos. Por ejemplo, si asignamos la regla del Listado 3-6 a los elementos **<section>** en lugar de los elementos **<p>**, como se muestra a continuación, el texto en el elemento **<p>** de nuestro documento se mostrará con un tamaño de 20 píxeles debido a que este elemento es hijo del elemento **<section>** y, por lo tanto, hereda sus estilos.

```
section {
  font-size: 20px;
}
```

Listado 3-7: *Heredando estilos*

Los estilos heredados de elementos en niveles superiores se pueden reemplazar por nuevos estilos definidos para los elementos en niveles inferiores de la jerarquía. Por ejemplo, podemos declarar una regla adicional para los elementos **<p>** que sobrescriba la propiedad **font-size** definida para el elemento **<section>** con un valor diferente.

```
section {
  font-size: 20px;
}
p {
  font-size: 36px;
}
```

Listado 3-8: *Sobrescribiendo estilos*

Ahora, el elemento **<p>** se nuestro documento se muestra con un tamaño de 36 píxeles porque el valor de la propiedad **font-size** asignada a los elementos **<section>** se modifica con la nueva regla asignada a los elementos **<p>**.

Mi texto

Figura 3-1: *Estilos en cascada*

Hágalo usted mismo: reemplace los estilos en su archivo misestilos.css por el código del Listado 3-8 y abra el documento del Listado 3-5 en su navegador. Debería ver el texto dentro del elemento **<p>** con un tamaño de 36 píxeles, tal como muestra la Figura 3-1.

3.2 Referencias

A veces es conveniente declarar todos los estilos en un archivo externo y luego cargar ese archivo desde cada documento que lo necesite, pero nos obliga a implementar diferentes mecanismos para establecer la relación entre las reglas CSS y los elementos dentro del documento que se verán afectados por las mismas.

Existen varios métodos para seleccionar los elementos que serán afectados por una regla CSS. En ejemplos anteriores hemos utilizado el nombre del elemento, pero también podemos

usar los valores de los atributos **id** y **class** para referenciar un solo elemento o un grupo de elementos, e incluso combinarlos para construir selectores más específicos.

Nombres

Una regla declarada con el nombre del elemento como selector afecta a todos los elementos de ese tipo encontrados en el documento. En ejemplos anteriores usamos el nombre **p** para modificar elementos **<p>**, pero podemos cambiar este nombre para trabajar con cualquier elemento en el documento que deseemos. Si en su lugar declaramos el nombre **span**, por ejemplo, se modificarán todos los textos dentro de elementos **<span>**.

```
span {
  font-size: 20px;
}
```

Listado 3-9: *Referenciando elementos* <span> *por su nombre*

Si queremos asignar los mismos estilos a elementos con nombres diferentes, podemos declarar los nombres separados por una coma. En el siguiente ejemplo, la regla afecta a todos los elementos **<p>** y **<span>** encontrados en el documento.

```
p, span {
  font-size: 20px;
}
```

Listado 3-10: *Declarando reglas con múltiples selectores*

También podemos referenciar solo elementos que se encuentran dentro de un elemento en particular listando los selectores separados por un espacio. Estos tipos de selectores se llaman *selectores de descendencia* porque afectan a elementos dentro de otros elementos, sin importar el lugar que ocupan en la jerarquía.

```
main p {
  font-size: 20px;
}
```

Listado 3-11: *Combinando selectores*

La regla en el Listado 3-11 afecta solo a los elementos **<p>** que se encuentran dentro de un elemento **<main>**, ya sea como contenido directo o insertados en otros elementos. Por ejemplo, el siguiente documento incluye un sección principal con una cabecera y una sección adicional. Ambos elementos incluyen elementos **<p>** para representar su contenido. Si aplicamos la regla del Listado 3-11 a este documento, el texto dentro de cada elemento **<p>** se mostrará con un tamaño de 20 píxeles porque todos son descendientes del elemento **<main>**.

```
<!DOCTYPE html>
<html lang="es">
```

```
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <link rel="stylesheet" href="misestilos.css">
</head>

<body>
  <main>
    <header>
      <h1>Título</h1>
      <p>Esta es la introducción</p>
    </header>
    <section>
      <p>Frase 1</p>
      <p>Frase 2</p>
      <p>Frase 3</p>
      <p>Frase 4</p>
    </section>
  </main>
</body>
</html>
```

Listado 3-12: *Probando selectores*

La regla del Listado 3-11 solo afecta a elementos **<p>** que se encuentran dentro del elemento **<main>**. Si, por ejemplo, agregamos un elemento **<footer>** al final del documento del Listado 3-12, los elementos **<p>** dentro de este pie de página no se verán modificados. La Figura 3-2 muestra lo que vemos cuando abrimos este documento en el navegador.

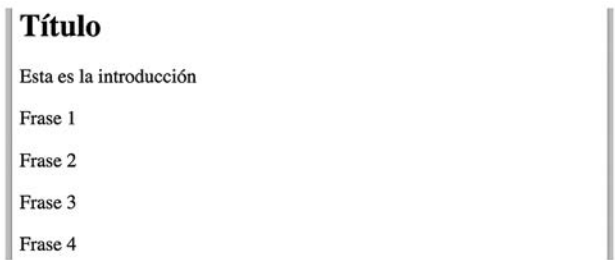

Figura 3-2: *Selectores de descendencia*

CSS incluye sintaxis adicionales para realizar una selección más precisa. Por ejemplo, podemos usar el carácter **>** para referenciar un elemento que es hijo directo de otro elemento.

```
section > p {
  font-size: 20px;
}
```

Listado 3-13: *Aplicando el selector >*

El carácter **>** indica que el elemento afectado es el elemento de la derecha cuando tiene como padre al elemento de la izquierda. La regla del Listado 3-13 modifica los elementos **<p>**

que son hijos de un elemento **<section>**. Cuando aplicamos esta regla al documento del Listado 3-12, el elemento **<p>** dentro del elemento **<header>** no se ve afectado.

Figura 3-3: *Selector de hijo*

Con el carácter **+** se crea otro selector que facilita CSS. Este selector referencia un elemento que está precedido por otro elemento. Ambos deben compartir el mismo elemento padre.

```
h1 + p {
  font-size: 20px;
}
```

Listado 3-14: *Aplicando el selector +*

La regla del Listado 3-14 afecta a todos los elementos **<p>** que se ubican inmediatamente después de un elemento **<h1>**. Si aplicamos esta regla a nuestro documento, solo se modificará el elemento **<p>** dentro de la cabecera porque es el único que está precedido por un elemento **<h1>**.

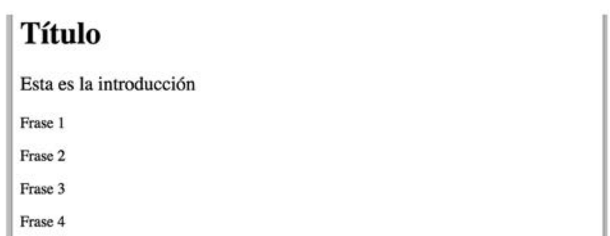

Figura 3-4: *Selector del elemento adyacente*

El selector anterior afecta solo al elemento **<p>** que se ubica inmediatamente después de un elemento **<h1>**. Si se coloca otro elemento entre ambos elementos, el elemento **<p>** no se modifica. CSS incluye el carácter ~ para crear un selector que afecta a todos los elementos que se ubican a continuación de otro elemento. Este selector es similar al anterior, pero el elemento afectado no tiene que encontrarse inmediatamente después del primer elemento. Esta regla afecta a todos los elementos encontrados, no solo al primero.

```
p ~ p {
  font-size: 20px;
}
```

Listado 3-15: *Aplicando el selector ~*

La regla del Listado 3-15 afecta a todos los elementos **<p>** que preceden a otro elemento **<p>**. En nuestro ejemplo, se modificarán todos los elementos **<p>** dentro del elemento **<section>**, excepto el primero, porque no existe un elemento **<p>** que preceda al primer elemento **<p>**.

Figura 3-5: *Selector general de hermano*

Hágalo usted mismo: cree un nuevo archivo HTML con el código del Listado 3-12. Cree un archivo CSS llamado misestilos.css y escriba dentro la regla que quiere probar. Inserte otros elementos entre los elementos **<p>** para ver cómo trabajan estas reglas.

Atributo Id

Las reglas anteriores afectan a los elementos del tipo indicado por el selector. Para seleccionar un elemento HTML sin considerar su tipo, podemos usar el atributo **id**. Este atributo es un nombre, un identificador exclusivo del elemento y, por lo tanto, lo podemos usar para encontrar un elemento en particular dentro del documento. Para referenciar un elemento usando su atributo **id**, el selector debe incluir el valor del atributo precedido por el carácter numeral (**#**).

```
#mitexto {
  font-size: 20px;
}
```

Listado 3-16: *Referenciando por medio del valor del atributo* id

La regla del Listado 3-16 solo se aplica al elemento identificado por el atributo **id** y el valor "mitexto", como el elemento **<p>** incluido en el siguiente documento.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <main>
    <section>
      <p>Frase 1</p>
      <p id="mitexto">Frase 2</p>
      <p>Frase 3</p>
```

```
      <p>Frase 4</p>
    </section>
  </main>
</body>
</html>
```

Listado 3-17: *Identificando un elemento* `<p>` *por medio de su atributo* `id`

La ventaja de este procedimiento es que cada vez que creamos una referencia usando el identificador **`mitexto`** en nuestro archivo CSS, solo se modifica el elemento con esa identificación, pero el resto de los elementos no se ven afectados. Esta es una forma muy específica de referenciar a un elemento y se usa comúnmente con elementos estructurales, como **`<section>`** o **`<div>`**. Debido a su especificidad, el atributo **`id`** también se usa frecuentemente para referenciar elementos desde JavaScript, tal como veremos en los próximos capítulos.

Atributo Class

En lugar de usar el atributo **`id`** para asignar estilos, en la mayoría de las ocasiones es mejor hacerlo con el atributo **`class`**. Este atributo es más flexible y se puede asignar a varios elementos dentro del mismo documento.

```
.mitexto {
  font-size: 20px;
}
```

Listado 3-18: *Referenciando por medio del valor del atributo* `class`

Para referenciar un elemento usando su atributo **`class`**, el selector debe incluir el valor del atributo precedido por un punto. Por ejemplo, la regla del Listado 3-18 afecta a todos los elementos que contienen un atributo **`class`** con el valor "mitexto", como en el siguiente ejemplo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <link rel="stylesheet" href="misestilos.css">
</head>

<body>
  <main>
    <section>
      <p class="mitexto">Frase 1</p>
      <p class="mitexto">Frase 2</p>
      <p>Frase 3</p>
      <p>Frase 4</p>
    </section>
  </main>
</body>
</html>
```

Listado 3-19: *Asignando estilos con el atributo* `class`

Los elementos **<p>** de las dos primeras líneas dentro de la sección principal del documento del Listado 3-19 incluyen el atributo **class** con el valor "mitexto". Debido a que se puede aplicar la misma regla a diferentes elementos del documento, estos dos primeros elementos son afectados por la regla del Listado 3-18. Por otro lado, los dos últimos elementos **<p>** no incluyen el atributo **class** y, por lo tanto, se mostrarán con los estilos por defecto.

Las reglas asignadas a través del atributo **class** se denominan *clases*. A un mismo elemento se le pueden asignar varias clases. Todo lo que tenemos que hacer es declarar los nombres de las clases separados por un espacio (por ejemplo, **class="texto1 texto2"**). Las clases también se pueden declarar como exclusivas para un tipo específico de elementos declarando el nombre del elemento antes del punto.

```
p.mitexto {
  font-size: 20px;
}
```

Listado 3-20: *Declarando una clase solo para elementos* <p>

En el Listado 3-20, hemos creado una regla que referencia la clase llamada **mitexto**, pero solo para los elementos **<p>**. Otros elementos que contengan el mismo valor en su atributo **class** no se verán afectados por esta regla.

Otros atributos

Aunque estos métodos de referencia cubren una amplia variedad de situaciones, a veces son insuficientes para encontrar el elemento exacto que queremos modificar. Para esas situaciones en las que los atributos **id** y **class** no son suficientes, CSS nos permite referenciar un elemento por medio de cualquier otro atributo que necesitemos. La sintaxis para definir esta clase de selectores incluye el nombre del elemento seguido del nombre del atributo en corchetes.

```
p[name] {
  font-size: 20px;
}
```

Listado 3-21: *Referenciando solo elementos* <p> *que tienen un atributo* name

La regla del Listado 3-21 modifica solo los elementos **<p>** que tienen un atributo llamado **name**. Para emular lo que hemos hecho con los atributos **id** y **class**, también podemos incluir el valor del atributo.

```
p[name="mitexto"] {
  font-size: 20px;
}
```

Listado 3-22: *Referenciando elementos* <p> *que tienen un atributo* name *con el valor* mitexto

CSS nos permite combinar el carácter = con otros caracteres para realizar una selección más específica. Los siguientes caracteres son los más utilizados.

```
p[name~="mi"] {
  font-size: 20px;
}
p[name^="mi"] {
  font-size: 20px;
}
p[name$="mi"] {
  font-size: 20px;
}
p[name*="mi"] {
  font-size: 20px;
}
```

Listado 3-23: *Aplicando selectores de atributo*

Las reglas del Listado 3-23 se han construido con selectores que incluyen los mismos atributos y valores, pero referencian diferentes elementos, como se describe a continuación.

- El selector con los caracteres **~=** referencia cualquier elemento **<p>** con un atributo **name** cuyo valor incluye la palabra "**mi**" (por ejemplo, **"mi texto"**, **"mi coche"**).
- El selector con los caracteres **^=** referencia cualquier elemento **<p>** con el atributo **name** cuyo valor comienza en "**mi**" (por ejemplo, **"mitexto"**, **"micoche"**).
- El selector con los caracteres **$=** referencia cualquier elemento **<p>** con un atributo **name** cuyo valor termina en "**mi**" (por ejemplo, **"textomi"**, **"cochemi"**).
- El selector con los caracteres ***=** referencia cualquier elemento **<p>** con un atributo **name** cuyo valor contiene la cadena de caracteres "**mi**" (en este caso, la cadena de caracteres podría también encontrarse en el medio del texto, como en **"textomicoche"**).

En estos ejemplos, usamos el elemento **<p>**, el atributo **name** y un texto arbitrario como "**mi**", pero se puede aplicar la misma técnica a cualquier atributo y valor que necesitemos.

Seudoclases

Las seudoclases son herramientas especiales de CSS que nos permiten referenciar elementos HTML por medio de sus características, como sus posiciones en el código o sus condiciones actuales. Las siguientes son las que más se utilizan.

:nth-child(valor**)**—Esta seudoclase selecciona un elemento de una lista de elementos hermanos que se encuentra en la posición especificada por el valor entre paréntesis.

:first-child—Esta seudoclase selecciona el primer elemento de una lista de elementos hermanos.

:last-child—Esta seudoclase selecciona el último elemento de una lista de elementos hermanos.

:only-child—Esta seudoclase selecciona un elemento cuando es el único hijo de otro elemento.

:first-of-type—Esta seudoclase selecciona el primer elemento de una lista de elementos del mismo tipo.

:not(selector**)**—Esta seudoclase selecciona los elementos que no coinciden con el selector entre paréntesis.

Con estos selectores, podemos realizar una selección más dinámica. En los siguientes ejemplos, vamos a aplicar algunos utilizando un documento sencillo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <main>
    <section>
      <p class="mitexto1">Frase 1</p>
      <p class="mitexto2">Frase 2</p>
      <p class="mitexto3">Frase 3</p>
      <p class="mitexto4">Frase 4</p>
    </section>
  </main>
</body>
</html>
```

Listado 3-24: *Creando un documento para probar las seudoclases*

La sección principal del documento del Listado 3-24 incluye cuatro elementos **`<p>`** que, considerando la estructura HTML, son hermanos y, por lo tanto, hijos del mismo elemento **`<section>`**. Usando seudoclases, podemos aprovechar esta organización y referenciar elementos sin importar cuánto conocemos acerca de sus atributos o valores. Por ejemplo, con la seudoclase **`:nth-child()`** podemos modificar todos los segundos elementos **`<p>`** que se encuentran en el documento.

```
p:nth-child(2) {
  font-size: 20px;
}
```

Listado 3-25: *Implementando la seudoclase* `:nth-child()`

Lo básico: las seudoclases se pueden agregar a cualquiera de los selectores que hemos mencionado con anterioridad. En la regla del Listado 3-25 hemos referenciado los elementos **`<p>`** usando el nombre **`p`**, pero esta regla también se podría haber escrito como **`.miclase:nth-child(2)`**, por ejemplo, para referenciar todo elemento que es hijo de otro elemento y que incluye el atributo **`class`** con el valor "miclase".

Lo que indica la seudoclase **:nth-child()** es algo como "el hijo en la posición…", por lo que el número entre paréntesis corresponde al número de posición del elemento hijo, o índice. Usando este tipo de referencias, podemos, por supuesto, seleccionar cualquier elemento hijo que queramos simplemente cambiando el número de índice. Por ejemplo, la siguiente regla afectará solo al último elemento **<p>** de nuestro documento.

```
p:nth-child(4) {
  font-size: 20px;
}
```

Listado 3-26: *Cambiando el índice para afectar a un elemento diferente*

Es posible asignar estilos a cada elemento creando una regla similar para cada uno de ellos.

```
p:nth-child(1) {
  background-color: #999999;
}
p:nth-child(2) {
  background-color: #CCCCCC;
}
p:nth-child(3) {
  background-color: #999999;
}
p:nth-child(4) {
  background-color: #CCCCCC;
}
```

Listado 3-27: *Generando una lista con la seudoclase* :nth-child()

En el Listado 3-27, hemos usado la seudoclase **:nth-child()** y la propiedad **background-color** para generar una lista de opciones que son claramente diferenciadas por el color de fondo. El valor **#999999** define un gris oscuro y el valor **#CCCCCC** define un gris claro. Por lo tanto, el primer elemento tendrá un fondo oscuro, el segundo un fondo claro, y así sucesivamente.

Figura 3-6: *La seudoclase* :nth-child()

Se podrían agregar más opciones a la lista incorporando nuevos elementos **<p>** en el código HTML y nuevas reglas con la seudoclase **:nth-child()** en la hoja de estilo. Sin embargo, esta técnica nos obligaría a extender el código innecesariamente y sería imposible aplicarla en sitios web con contenido dinámico. Una alternativa es la que ofrecen las palabras clave **odd** y **even** disponibles para esta seudoclase.

```
p:nth-child(odd) {
  background-color: #999999;
}
p:nth-child(even) {
  background-color: #CCCCCC;
}
```

Listado 3-28: *Implementando las palabras clave* `odd` *y* `even`

La palabra clave **`odd`** para la seudoclase **`:nth-child()`** afecta a los elementos **`<p>`** que son hijos de otro elemento y tienen un índice impar, y la palabra clave **`even`** afecta a aquellos que tienen un índice par. Usando estas palabras clave, solo necesitamos dos reglas para configurar la lista completa, sin importar lo larga que sea. Incluso cuantas más opciones o filas se incorporen más adelante al documento, menos necesario será agregar otras reglas al archivo CSS. Los estilos se asignarán automáticamente a cada elemento de acuerdo con su posición.

Además de esta seudoclase existen otras que están relacionadas y también nos pueden ayudar a encontrar elementos específicos dentro de la jerarquía, como **`:first-child`**, **`:last-child`** y **`:only-child`**. La seudoclase **`:first-child`** referencia solo el primer elemento hijo, **`:last-child`** referencia solo el último elemento hijo, y **`:only-child`** afecta a un elemento cuando es el único elemento hijo. Estas seudoclases no requieren palabras clave o ningún parámetro adicional y se implementan según muestra el siguiente ejemplo.

```
p:last-child {
  font-size: 20px;
}
```

Listado 3-29: *Usando* `:last-child` *para modificar el último elemento* `<p>` *de la lista*

Otra seudoclase importante es **`:not()`**. Con esta seudoclase podemos seleccionar elementos que no coinciden con un selector. Por ejemplo, la siguiente regla asigna un margen de 0 píxeles a todos los elementos con nombres diferentes de **`p`**.

```
:not(p) {
  margin: 0px;
}
```

Listado 3-30: *Aplicando estilos a todos los elementos excepto* `<p>`

En lugar del nombre del elemento podemos usar cualquier otro selector que necesitemos. En el siguiente listado, se verán afectados todos los elementos dentro del elemento **`<section>`** excepto aquellos que contengan el atributo **`class`** con el valor "mitexto2".

```
section :not(.mitexto2) {
  margin: 0px;
}
```

Listado 3-31: *Definiendo una excepción por medio del atributo* `class`

Cuando aplicamos esta última regla al código HTML del Listado 3-24, el navegador asigna los estilos por defecto al elemento **<p>** con la clase **mitexto2** y asigna un margen de 0 píxeles al resto (el elemento **<p>** con la clase **mitexto2** es el que no se verá afectado por la regla). El resultado se ilustra en la Figura 3-7. Los elementos **<p>** primero, tercero y cuarto se presentan sin margen, pero el segundo elemento **<p>** se muestra con los márgenes superiores e inferiores por defecto.

Figura 3-7: *La seudoclase* **:**not()

3.3 Propiedades

Las propiedades son la pieza central de CSS. Todos los estilos que podemos aplicar a un elemento se definen por medio de propiedades. Ya hemos introducido algunas en los ejemplos anteriores, pero hay cientos de propiedades disponibles. Para simplificar su estudio, se pueden clasificar en dos tipos: propiedades de formato y propiedades de diseño. Las propiedades de formato se encargan de dar forma a los elementos y su contenido, mientras que las de diseño están enfocadas a determinar el tamaño y la posición de los elementos en la pantalla. Así mismo, las propiedades de formato se pueden clasificar según el tipo de modificación que producen. Por ejemplo, algunas propiedades cambian el tipo de letra que se usa para mostrar el texto, otras generan un borde alrededor del elemento, asignan un color de fondo, etc. En este capítulo vamos a introducir las propiedades de formato siguiendo esta clasificación.

Texto

Desde CSS se pueden controlar varios aspectos del texto, como el tipo de letra que se usa para mostrar en pantalla, el espacio entre líneas, la alineación, etc. Las siguientes son las propiedades disponibles para definir el tipo de letra, tamaño, y estilo de un texto.

font-family—Esta propiedad declara el tipo de letra que se usa para mostrar el texto. Se pueden declarar múltiples valores separados por coma para ofrecer al navegador varias alternativas en caso de que algunos tipos de letra no se encuentren disponibles en el ordenador del usuario. Algunos de los valores estándar son **Georgia**, **"Times New Roman"**, **Arial**, **Helvetica**, **"Arial Black"**, **Gadget**, **Tahoma**, **Geneva**, **Helvetica**, **Verdana**, **Geneva**, **Impact**, y **sans-serif** (los nombres compuestos por más de una palabra se deben declarar entre comillas dobles).

font-size—Esta propiedad determina el tamaño de la letra. El valor puede ser declarado en píxeles (**px**), porcentaje (**%**), o usando cualquiera de las unidades disponibles en CSS como **em**, **rem**, **pt**, etc. El valor por defecto es normalmente **16px**.

font-weight—Esta propiedad determina si el texto se mostrará en negrita o no. Los valores disponibles son **normal** y **bold**, pero también podemos asignar los valores **100**, **200**, **300**, **400**, **500**, **600**, **700**, **800**, y **900** para determinar el grosor de la letra (solo disponibles para algunos tipos de letra).

font-style—Esta propiedad determina el estilo de la letra. Los valores disponibles son **`normal`**, **`italic`**, y **`oblique`**.

font—Esta propiedad nos permite declarar múltiples valores al mismo tiempo. Los valores deben declararse separados por un espacio y en un orden preciso. El estilo y el grosor se deben declarar antes que el tamaño, y el tipo de letra al final (por ejemplo, **`font: bold 24px Arial,sans-serif`**).

Estas propiedades se deben aplicar a cada elemento (o elemento padre) cuyo texto queremos modificar. Por ejemplo, el siguiente documento incluye una cabecera con un título y una sección con un párrafo. Como queremos asignar diferentes tipos de letra al título y al texto, tenemos que incluir el atributo **`id`** en cada elemento para poder identificarlos desde las reglas CSS.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <header>
    <span id="titulo">Hojas de Estilo en Cascada</span>
  </header>
  <section>
    <p id="descripcion">Hojas de estilo en cascada (o CSS, siglas en
inglés de Cascading Style Sheets) es un lenguaje de diseño gráfico para
definir y crear la presentación de un documento estructurado escrito en
HTML.</p>
  </section>
  <footer>
    <a href="http://www.jdgauchat.com">www.jdgauchat.com</a>
  </footer>
</body>
</html>
```

Listado 3-32: *Probando propiedades de formato*

Las reglas deben incluir todas las propiedades requeridas para asignar los estilos que deseamos. Por ejemplo, si queremos asignar un tipo de letra y un nuevo tamaño al texto, tenemos que incluir las propiedades **`font-family`** y **`font-size`**.

```
#titulo {
  font-family: Verdana, sans-serif;
  font-size: 26px;
}
```

Listado 3-33: *Cambiando el tipo de letra y el tamaño del título*

Aunque podemos declarar varias propiedades en la misma regla para modificar diferentes aspectos del elemento, CSS ofrece una propiedad abreviada llamada **`font`** que nos permite

definir todas las características del texto en una sola línea de código. Los valores asignados a esta propiedad se deben declarar separados por un espacio. Por ejemplo, la siguiente regla asigna el estilo **`bold`** (negrita), un tamaño de 26 píxeles, y el tipo de letra **`Verdana`** al elemento **`titulo`**.

```
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-34: *Cambiando el tipo de letra con la propiedad* `font`

Cuando la regla del Listado 3-34 se aplica al documento del Listado 3-32, el título se muestra con los valores definidos por la propiedad **`font`** y el párrafo se presenta con los estilos por defecto.

Hojas de Estilo en Cascada

Hojas de estilo en cascada (o CSS, siglas en inglés de Cascading Style Sheets) es un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en HTML.

www.jdgauchat.com

Figura 3-8: *La propiedad* `font`

Hágalo usted mismo: cree un nuevo archivo HTML con el código del Listado 3-32. Cree o actualice el archivo misestilos.css con la regla del Listado 3-34. Abra el documento en su navegador. Debería ver algo parecido a lo que se muestra en la Figura 3-8.

Desde CSS podemos cambiar no solo el tipo de letra, sino también otros aspectos del texto, como el alineamiento, la sangría, el espacio entre líneas, etc. Las siguientes son algunas de las propiedades disponibles para este propósito.

text-align—Esta propiedad alinea el texto dentro de un elemento. Los valores disponibles son **`left`**, **`right`**, **`center`**, y **`justify`**.

text-align-last—Esta propiedad alinea la última línea de un párrafo. Los valores disponibles son **`left`**, **`right`**, **`center`**, y **`justify`**.

text-indent—Esta propiedad define el tamaño de la sangría de un párrafo (el espacio vacío al comienzo de la línea). El valor se puede declarar en píxeles (**`px`**), porcentaje (**`%`**), o usando cualquiera de las unidades disponibles en CSS, como **`em`**, **`rem`**, **`pt`**, etc.

letter-spacing—Esta propiedad define el espacio entre letras. El valor se debe declarar en píxeles (**`px`**), porcentaje (**`%`**) o usando cualquiera de las unidades disponibles en CSS, como **`em`**, **`rem`**, **`pt`**, etc.

word-spacing—Esta propiedad define el ancho del espacio entre palabras. El valor puede ser declarado en píxeles (**`px`**), porcentaje (**`%`**) o usando cualquiera de las unidades disponibles en CSS, como **`em`**, **`rem`**, **`pt`**, etc.

line-height—Esta propiedad define el espacio entre líneas. El valor se puede declarar en píxeles (`px`), porcentaje (`%`) o usando cualquiera de las unidades disponibles en CSS, como `em`, `rem`, `pt`, etc.

vertical-align—Esta propiedad alinea elementos verticalmente. Se usa frecuentemente para alinear texto con imágenes (la propiedad se aplica a la imagen). Los valores disponibles son `baseline`, `sub`, `super`, `text-top`, `text-bottom`, `middle`, `top`, y `bottom`.

Por defecto, los navegadores alinean el texto a la izquierda, pero podemos cambiar la alineación con la propiedad `text-align`. La siguiente regla centra el párrafo de nuestro documento.

```
#titulo {
  font: bold 26px Verdana, sans-serif;
}
#descripcion {
  text-align: center;
}
```

Listado 3-35: *Alineando el texto con la propiedad* `text-align`

Figura 3-9: *La propiedad* `text-align`

El resto de las propiedades listadas arriba son simples de aplicar. Por ejemplo, podemos definir el tamaño del espacio entre palabras con la propiedad `word-spacing`.

```
#titulo {
  font: bold 26px Verdana, sans-serif;
}
#descripcion {
  text-align: center;
  word-spacing: 20px;
}
```

Listado 3-36: *Definiendo el espacio entre palabras con la propiedad* `word-spacing`

Figura 3-10: *La propiedad* `word-spacing`

CSS también ofrece la siguiente propiedad para decorar el texto con una línea.

text-decoration—Esta propiedad resalta el texto con una línea. Los valores disponibles son **`underline`**, **`overline`**, **`line-through`**, y **`none`**.

La propiedad **`text-decoration`** es particularmente útil con enlaces. Por defecto, los navegadores muestran los enlaces subrayados. Si queremos eliminar la línea, podemos declarar esta propiedad con el valor **`none`**. El siguiente ejemplo agrega una regla a nuestra hoja de estilo para modificar los elementos **`<a>`** del documento.

```
#titulo {
  font: bold 26px Verdana, sans-serif;
}
#descripcion {
  text-align: center;
}
a {
  text-decoration: none;
}
```

Listado 3-37: *Eliminando las líneas en los enlaces de nuestro documento*

Figura 3-11: *La propiedad* `text-decoration`

Hasta el momento, hemos utilizado el tipo de letra **`Verdana`** (y la alternativa **`sans-serif`** en caso de que **`Verdana`** no esté disponible). Este tipo de letra es parte de un grupo conocido como *fuentes seguras* porque se encuentran disponibles en la mayoría de los ordenadores y, por lo tanto, podemos usarla con seguridad. El problema con las fuentes en la Web es que los navegadores no las descargan desde el servidor del sitio web, las cargan desde el ordenador del usuario, y los usuarios tienen diferentes tipos de letra instaladas en sus sistemas. Si usamos una fuente que el usuario no posee, el diseño de nuestro sitio web se verá diferente. Usando fuentes seguras nos aseguramos de que nuestro sitio web se verá de igual manera en cualquier navegador u ordenador, pero el problema es que los tipos de letras incluidos en este grupo son pocos. Para ofrecer una solución y permitir plena creatividad al diseñador, CSS incluye la regla **`@font-face`**. La regla **`@font-face`** es una regla reservada que permite a los diseñadores incluir un archivo con la fuente de letra a usar para mostrar el texto de una página web. Si la usamos, podemos incluir cualquier fuente que deseemos en nuestro sitio web con solo facilitar el archivo que la contiene.

La regla **`@font-face`** necesita al menos dos propiedades para declarar la fuente y cargar el archivo. La propiedad **`font-family`** especifica el nombre que queremos usar para referenciar este tipo de letra y la propiedad **`src`** indica la URL del archivo con las especificaciones de la fuente (la sintaxis de la propiedad **`src`** requiere el uso de la función **`url()`** para indicar la URL del archivo). En el siguiente ejemplo, el nombre **`MiNuevaLetra`** se asigna a nuestra fuente y el archivo **`font.ttf`** se indica como el archivo a cargar.

```
#titulo {
  font: bold 26px MiNuevaLetra, Verdana, sans-serif;
}
@font-face {
  font-family: "MiNuevaLetra";
  src: url("font.ttf");
}
```

Listado 3-38: *Cargando un tipo de letra personalizado para el título*

Una vez que se carga la fuente, podemos usarla en cualquier elemento del documento por medio de su nombre (**`MiNuevaLetra`**). En la propiedad **`font`** de la regla **`#titulo`** del Listado 3-38, especificamos que el título se mostrará con la nueva fuente o con las fuentes alternativas **`Verdana`** y **`sans-serif`** si nuestra fuente no se ha cargado.

Hojas de Estilo en Cascada

Hojas de estilo en cascada (o CSS, siglas en inglés de Cascading Style Sheets) es un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en HTML.

www.jdgauchat.com

Figura 3-12: *Título con un tipo de letra personalizado*

Hágalo usted mismo: descargue el archivo font.ttf desde nuestro sitio web. Copie el archivo dentro del directorio de su proyecto. Actualice su hoja de estilo con el código del Listado 3-38 y abra el documento del Listado 3-32 en su navegador. Puede encontrar más fuentes como la que se muestra en la Figura 3-12 en www.moorstation.org/typoasis/designers/steffmann/.

Lo básico: una función es un trozo de código que realiza una tarea específica y devuelve el resultado. La función `url()`, por ejemplo, tiene la tarea de cargar el archivo indicado entre paréntesis y devolver su contenido. CSS incluye varias funciones para generar los valores de sus propiedades. Introduciremos algunas de estas funciones a continuación y aprenderemos más sobre funciones en el Capítulo 6.

Colores

Existen dos formas de declarar un color en CSS: podemos usar una combinación de tres colores básicos (rojo, verde y azul), o definir el matiz, la saturación y la luminosidad. El color final se crea considerando los niveles que asignamos a cada componente. Dependiendo del tipo de sistema que utilizamos para definir el color, tendremos que declarar los niveles usando números hexadecimales (desde 00 a FF), números decimales (desde 0 a 255) o porcentajes. Por ejemplo, si decidimos usar una combinación de niveles de rojo, verde y azul, podemos declarar los niveles con números hexadecimales. En este caso, los valores del color se declaran en secuencia y precedidos por el carácter numeral, como en #996633 (99 es el nivel de rojo, 66

es el nivel de verde y 33 es el nivel de azul). Para definir colores con otros tipos de valores, CSS ofrece las siguientes funciones.

rgb(rojo, verde, azul**)**—Esta función define un color por medio de los valores especificados por los atributos (desde 0 a 255). El primer valor representa el nivel de rojo, el segundo valor representa el nivel de verde y el último valor el nivel de azul (por ejemplo, `rgb(153, 102, 51)`).

rgba(rojo, verde, azul, alfa**)**—Esta función es similar a la función `rgb()`, pero incluye un componente adicional para definir la opacidad (alfa). El valor se puede declarar entre 0 y 1, con 0 como totalmente transparente y 1 como totalmente opaco.

hsl(matiz, saturación, luminosidad**)**—Esta función define un color desde los valores especificados por los atributos. Los valores se declaran en números decimales y porcentajes.

hsla(matiz, saturación, luminosidad, alfa**)**—Esta función es similar a la función `hsl()`, pero incluye un componente adicional para definir la opacidad (alfa). El valor se puede declarar entre 0 y 1, con 0 como totalmente transparente y 1 como totalmente opaco.

Como veremos más adelante, son varias las propiedades que requieren valores que definen colores, pero la siguiente es la que se utiliza con más frecuencia:

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

La siguiente regla asigna un gris claro al título de nuestro documento usando números hexadecimales.

```
#titulo {
  font: bold 26px Verdana, sans-serif;
  color: #CCCCCC;
}
```

Listado 3-39: *Asignando un color al título*

Cuando los niveles de rojo, verde y azul son iguales, como en este caso, el color final se encuentra dentro de una escala de grises, desde negro (#000000) a blanco (#FFFFFF).

Hojas de Estilo en Cascada

Hojas de estilo en cascada (o CSS, siglas en inglés de Cascading Style Sheets) es un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en HTML.

www.jdgauchat.com

Figura 3-13: *Título con un color personalizado*

Declarar un color con una función es bastante parecido: solo tenemos que reemplazar el valor hexadecimal por la función que queremos utilizar. Por ejemplo, podemos definir el mismo color de grises con la función `rgb()` y valores decimales.

```
#titulo {
  font: bold 26px Verdana, sans-serif;
  color: rgb(204, 204, 204);
}
```

Listado 3-40: *Asignando un color con la función* `rgb()`

La función **`hsl()`** es simplemente otra función disponible para generar un color, pero es más intuitiva que **`rgb()`**. A algunos diseñadores les resulta más fácil crear grupos de colores usando **`hsl()`**. Los valores requeridos por esta función definen el matiz, la saturación y la luminosidad. El matiz es un color extraído de una rueda imaginaria, expresado en grados desde 0 a 360: alrededor de 0 y 360 se encuentran los rojos, cerca de 120 los verdes, y cerca de 240 los azules. La saturación se representa en porcentaje, desde 0 % (escala de grises) a 100 % (todo color o totalmente saturado), y la luminosidad es también un valor en porcentaje, desde 0 % (completamente negro) a 100 % (completamente blanco); un valor de 50 % representa una luminosidad promedio. Por ejemplo, en la siguiente regla, la saturación se define como 0 % para crear un color dentro de la escala de grises y la luminosidad se especifica en 80 % para obtener un gris claro.

```
#titulo {
  font: bold 26px Verdana, sans-serif;
  color: hsl(0, 0%, 80%);
}
```

Listado 3-41: *Asignando un color con la función* `hsl()`

IMPORTANTE: CSS define una propiedad llamada **`opacity`** para declarar la opacidad de un elemento. Esta propiedad presenta el problema de que el valor de opacidad para un elemento lo heredan sus elementos hijos. Ese problema se resuelve con las funciones **`rgba()`** y **`hsla()`**, con las que podemos asignar un valor de opacidad al fondo de un elemento, como veremos a continuación, sin que su contenido se vea afectado.

Lo básico: no es práctico encontrar los colores adecuados para nuestro sitio web combinando números y valores. Para facilitar esta tarea, los ordenadores personales incluyen varias herramientas visuales que nos permiten seleccionar un color y obtener su valor. Además, la mayoría de los editores de fotografía y programas gráficos disponibles en el mercado incluyen una herramienta que muestra una paleta desde donde podemos seleccionar un color y obtener el correspondiente valor hexadecimal o decimal. Para encontrar los colores adecuados, puede usar estas herramientas o cualquiera de las disponibles en la Web, como **www.colorhexa.com** o **htmlcolorcodes.com**.

Tamaño

Por defecto, el tamaño de la mayoría de los elementos se determina según el espacio disponible en el contenedor. El ancho de un elemento se define como 100 %, lo cual significa que será tan ancho como su contenedor, y tendrá una altura determinada por su contenido.

Esta es la razón por la que el elemento **<p>** del documento del Listado 3-32 se extendía a los lados de la ventana del navegador y no era más alto de lo necesario para contener las líneas del párrafo. CSS define las siguientes propiedades para declarar un tamaño personalizado:

width—Esta propiedad declara el ancho de un elemento. El valor se puede especificar en píxeles, porcentaje, o con la palabra clave **auto** (por defecto). Cuando el valor se especifica en porcentaje, el ancho se calcula según el navegador a partir del ancho del contenedor, y cuando se declara con el valor **auto**, el elemento se expande hasta ocupar todo el espacio horizontal disponible dentro del contenedor.

height—Esta propiedad declara la altura de un elemento. El valor se puede especificar en píxeles, porcentaje, o con la palabra clave **auto** (por defecto). Cuando el valor se especifica en porcentaje, el navegador calcula la altura a partir de la altura del contenedor, y cuando se declara con el valor **auto**, el elemento adopta la altura de su contenedor.

Los navegadores generan una caja alrededor de cada elemento que determina el área que ocupa en la pantalla. Cuando declaramos un tamaño personalizado, la caja se modifica y el contenido del elemento se adapta para encajar dentro de la nueva área, tal como muestra la Figura 3-14.

Figura 3-14: *Caja personalizada*

Por ejemplo, si declaramos un ancho de 200 píxeles para el elemento **<p>** de nuestro documento, las líneas del texto serán menos largas, pero se agregarán nuevas líneas para mostrar todo el párrafo.

```
#titulo {
  font: bold 26px Verdana, sans-serif;
}
#descripcion {
  width: 200px;
}
```

Listado 3-42: *Asignando un ancho personalizado*

Hojas de Estilo en Cascada

Hojas de estilo en cascada (o CSS, siglas en inglés de Cascading Style Sheets) es un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en HTML.

www.jdgauchat.com

Figura 3-15: *Contenido principal con un tamaño personalizado*

La regla del Listado 3-42 declara el ancho del elemento **<p>**, pero la altura queda aún determinada por su contenido, lo que significa que la caja generada por este elemento será más alta para contener el párrafo completo, según ilustra la Figura 3-15. Si también queremos restringir la altura del elemento, podemos usar la propiedad **height**. La siguiente regla reduce la altura del elemento **<p>** de nuestro documento a 100 píxeles.

```
#titulo {
  font: bold 26px Verdana, sans-serif;
}
#descripcion {
  width: 200px;
  height: 100px;
}
```

Listado 3-43: *Asignando una altura personalizada*

El problema con elementos que tienen un tamaño definido es que a veces el contenido no se puede mostrar en su totalidad. Por defecto, los navegadores muestran el resto del contenido fuera del área de la caja. En consecuencia, parte del contenido de una caja con tamaño personalizado se puede posicionar sobre el contenido del elemento que se encuentra debajo, tal como se ilustra en la Figura 3-16.

Hojas de Estilo en Cascada

Hojas de estilo en cascada (o CSS, siglas en inglés de Cascading Style Sheets) es un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en HTML.
www.jdgauchat.com

Figura 3-16: *Contenido desbordado*

En nuestro ejemplo, el texto que se encuentra fuera de la caja determinada por el elemento **<p>** ocupa el espacio correspondiente al elemento **<footer>** y, por lo tanto, el contenido de ambos elementos se encuentra superpuesto. CSS incluye las siguientes propiedades para resolver este problema:

overflow—Esta propiedad especifica cómo se mostrará el contenido que desborda el elemento. Los valores disponibles son **visible** (por defecto), **hidden** (esconde el contenido que no entra dentro de la caja), **scroll** (muestra barras laterales para desplazar el contenido), **auto** (deja que el navegador decida qué hacer con el contenido).

overflow-x—Esta propiedad especifica cómo se mostrará el contenido que desborda el elemento horizontalmente. Acepta los mismos valores que la propiedad **overflow**.

overflow-y—Esta propiedad especifica cómo se mostrará el contenido que desborda el elemento verticalmente. Acepta los mismos valores que la propiedad **overflow**.

overflow-wrap—Esta propiedad indica si un palabra debería ser dividida en un punto arbitrario cuando no hay suficiente espacio para mostrarla en la línea. Los valores disponibles son **normal** (la línea será dividida naturalmente) y **break-word** (las palabras se dividirán en puntos arbitrarios para acomodar la línea de texto en el espacio disponible).

Con estas propiedades podemos determinar cómo se mostrará el contenido cuando no hay suficiente espacio disponible. Por ejemplo, podemos ocultar el texto que desborda el elemento asignando el valor **hidden** a la propiedad **overflow**.

```
#titulo {
  font: bold 26px Verdana, sans-serif;
}
#descripcion {
  width: 200px;
  height: 100px;
  overflow: hidden;
}
```

Listado 3-44: *Ocultando el contenido que desborda el elemento*

Figura 3-17: *Contenido ocultado*

Si queremos que el usuario pueda ver el texto que se ha ocultado, podemos asignar el valor **scroll** y forzar al navegador a mostrar barras laterales para desplazar el contenido.

```
#titulo {
  font: bold 26px Verdana, sans-serif;
}
#descripcion {
  width: 200px;
  height: 100px;
  overflow: scroll;
}
```

Listado 3-45: *Incorporando barras de desplazamiento*

Figura 3-18: *Barras de desplazamiento*

El tamaño del elemento no queda solo determinado por el ancho y la altura de su caja, sino también por el relleno y los márgenes. CSS nos permite designar espacio alrededor de la caja

para separar el elemento de otros elementos a su alrededor (margen), además de incluir espacio entre los límites de la caja y su contenido (relleno). La Figura 3-19 ilustra cómo se aplican estos espacios a un elemento.

Figura 3-19: *Márgenes, rellenos y bordes*

CSS incluye las siguientes propiedades para definir márgenes y rellenos para un elemento.

margin—Esta propiedad declara el margen de un elemento. El margen es el espacio que hay alrededor de la caja. Puede recibir cuatro valores que representan el margen superior, derecho, inferior, e izquierdo, en ese orden y separados por un espacio (por ejemplo, **`margin: 10px 30px 10px 30px;`**). Sin embargo, si solo se declaran uno, dos o tres valores, los otros toman los mismos valores (por ejemplo, **`margin: 10px 30px`** asigna 10 píxeles al margen superior e inferior y 30 píxeles al margen izquierdo y derecho). Los valores se pueden declarar independientemente usando las propiedades asociadas **`margin-top`**, **`margin-right`**, **`margin-bottom`** y **`margin-left`** (por ejemplo, **`margin-left: 10px;`**). La propiedad también acepta el valor **`auto`** para obligar al navegador a calcular el margen (usado para centrar un elemento dentro de su contenedor).

padding—Esta propiedad declara el relleno de un elemento. El relleno es el espacio entre el contenido del elemento y los límites de su caja. Los valores se declaran de la misma forma que lo hacemos para la propiedad **`margin`**, aunque también se pueden declarar de forma independiente con las propiedades **`padding-top`**, **`padding-right`**, **`padding-bottom`** y **`padding-left`** (por ejemplo, **`padding-top: 10px;`**).

La siguiente regla agrega márgenes y relleno a la cabecera de nuestro documento. Debido a que asignamos solo un valor, el mismo valor se usa para definir todos los márgenes y rellenos del elemento (superior, derecho, inferior e izquierdo, en ese orden).

```
header {
  margin: 30px;
  padding: 15px;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-46: *Agregando márgenes y relleno*

El tamaño de un elemento y sus márgenes se agregan para calcular el área que ocupa. Lo mismo pasa con el relleno y el borde (estudiaremos bordes más adelante). El tamaño final de un elemento se calcula con la fórmula: tamaño + márgenes + relleno + bordes. Por ejemplo, si tenemos un elemento con un ancho de 200 píxeles y un margen de 10 píxeles a cada lado, el

ancho del área ocupada por el elemento será de 220 píxeles. El total de 20 píxeles de margen se agrega a los 200 píxeles del elemento y el valor final se representa en la pantalla.

Hojas de Estilo en Cascada

Figura 3-20: *Cabecera con márgenes y relleno personalizados*

Hágalo usted mismo: reemplace las reglas en su archivo CSS con las reglas del Listado 3-46 y abra el documento en su navegador. Debería ver algo similar a lo que se representa en la Figura 3-20. Cambie el valor de la propiedad **`margin`** para ver cómo afecta a los márgenes.

IMPORTANTE: como veremos más adelante en este capítulo, los elementos se clasifican en dos tipos principales: Block (bloque) e Inline (en línea). Los elementos Block pueden tener un tamaño personalizado, pero los elementos Inline solo pueden ocupar el espacio determinado por sus contenidos. El elemento **<span>** que usamos para definir el título de la cabecera se declara por defecto como elemento Inline y, por lo tanto, no puede tener un tamaño y unos márgenes personalizados. Esta es la razón por la que en nuestro ejemplo asignamos márgenes y relleno al elemento **<header>** en lugar de al elemento **<span>** (los elementos estructurales se definen todos como elementos Block).

Fondo

Los elementos pueden incluir un fondo que se muestra detrás del contenido del elemento y a través del área ocupada por el contenido y el relleno. Debido a que el fondo puede estar compuesto por colores e imágenes, CSS define varias propiedades para generarlo.

background-color—Esta propiedad asigna un fondo de color a un elemento.

background-image—Esta propiedad asigna una o varias imágenes al fondo de un elemento. La URL del archivo se declara con la función **`url()`** (por ejemplo, **`url("ladrillos.jpg")`**). Si se requiere más de una imagen, los valores se deben separar por una coma.

background-position—Esta propiedad declara la posición de comienzo de una imagen de fondo. Los valores se pueden especificar en porcentaje, píxeles o usando una combinación de las palabras clave **`center`**, **`left`**, **`right`**, **`top`**, y **`bottom`**.

background-size—Esta propiedad declara el tamaño de la imagen de fondo. Los valores se pueden especificar en porcentaje, píxeles, o usando las palabras clave **`cover`** y **`contain`**. La palabra clave **`cover`** expande la imagen hasta que su ancho o su altura cubren el área del elemento, mientras que **`contain`** estira la imagen para ocupar toda el área del elemento.

background-repeat—Esta propiedad determina cómo se distribuye la imagen de fondo usando cuatro palabras clave: **`repeat`**, **`repeat-x`**, **`repeat-y`** y **`no-repeat`**. La palabra clave **`repeat`** repite la imagen en el eje vertical y horizontal, mientras que **`repeat-x`** y

repeat-y lo hacen solo en el eje horizontal o vertical, respectivamente. Finalmente, **no-repeat** muestra la imagen de fondo una sola vez.

background-origin—Esta propiedad determina si la imagen de fondo se posicionará considerando el borde, el relleno o el contenido del área del elemento. Los valores disponibles son **border-box**, **padding-box**, y **content-box**.

background-clip—Esta propiedad declara el área a cubrir por el fondo usando los valores **border-box**, **padding-box**, y **content-box**. El primer valor corta la imagen al borde de la caja del elemento, el segundo corta la imagen en el relleno de la caja y el tercero corta la imagen alrededor del contenido de la caja.

background-attachment—Esta propiedad determina si la imagen es estática o se desplaza con el resto de los elementos usando dos valores: **scroll** (por defecto) y **fixed**. El valor **scroll** hace que la imagen se desplace con la página, y el valor **fixed** fija la imagen de fondo en su lugar original.

background—Esta propiedad nos permite declarar todos los atributos del fondo al mismo tiempo.

Los fondos más comunes se crean con colores. La siguiente regla implementa la propiedad **background-color** para agregar un fondo gris a la cabecera de nuestro documento.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  background-color: #CCCCCC;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-47: *Agregando un color de fondo*

Debido a que definimos márgenes de 30 píxeles a los lados, la cabecera queda centrada en la ventana del navegador, pero el texto dentro del elemento **<header>** aún se encuentra alineado a la izquierda (la alineación por defecto). En el ejemplo del Listado 3-47, además de cambiar el fondo, también incluimos la propiedad **text-align** para centrar el título. El resultado se muestra en la Figura 3-21.

Figura 3-21: *Fondo de color*

Además de colores, también podemos usar imágenes de fondo. En este caso, tenemos que declarar el fondo con la propiedad **background-image** y declarar la URL de la imagen con la función **url()**, como lo hacemos en el siguiente ejemplo.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  background-image: url("ladrillosclaros.jpg");
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-48: *Agregando una imagen de fondo*

El problema con las imágenes es que no siempre son del mismo tamaño que la caja creada por el elemento. Por esta razón, los navegadores repiten la imagen una y otra vez hasta cubrir toda el área. El resultado se muestra en la Figura 3-22.

Figura 3-22: *Imagen de fondo*

Hágalo usted mismo: reemplace las reglas en su archivo CSS por las reglas del Listado 3-48. Descargue la imagen ladrillosclaros.jpg desde nuestro sitio web. Copie la imagen en el mismo directorio donde se encuentra su documento. Abra el documento en su navegador. Debería ver algo parecido a lo que se muestra en la Figura 3-22.

Si queremos modificar el comportamiento por defecto, podemos usar el resto de las propiedades de fondo disponibles. Por ejemplo, si asignamos el valor **`repeat-y`** a la propiedad **`background-repeat`**, el navegador solo repetirá la imagen en el eje vertical.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  background-image: url("ladrillosclaros.jpg");
  background-repeat: repeat-y;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-49: *Configurando el fondo*

Figura 3-23: *Fondo de imagen*

Cuando nuestro diseño requiere varios valores para configurar el fondo, podemos declararlos todos juntos con la propiedad **background**. Esta propiedad nos permite declarar diferentes aspectos del fondo en una sola línea de código.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  background: #CCCCCC url("ladrillosclaros.jpg") repeat-y;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-50: *Configurando el fondo con la propiedad* `background`

La regla del Listado 3-50 especifica los mismos valores que usamos anteriormente, pero ahora combina una imagen de fondo con un color. El resultado se muestra en la Figura 3-24.

Figura 3-24: *Imagen de fondo combinada con un color de fondo*

Bordes

Los elementos pueden incluir un borde en los límites de la caja del elemento. Por defecto, los navegadores no muestran ningún borde, pero podemos usar las siguientes propiedades para definirlo.

border-width—Esta propiedad define el ancho del borde. Acepta hasta cuatro valores separados por un espacio para especificar el ancho de cada lado del borde (superior, derecho, inferior, e izquierdo, es ese orden). También podemos declarar el ancho para cada lado de forma independiente con las propiedades **border-top-width**, **border-bottom-width**, **border-left-width**, y **border-right-width**.

border-style—Esta propiedad define el estilo del borde. Acepta hasta cuatro valores separados por un espacio para especificar los estilos de cada lado del borde (superior, derecho, inferior, e izquierdo, en ese orden). Los valores disponibles son **none**, **hidden**, **dotted**, **dashed**, **solid**, **double**, **groove**, **ridge**, **inset**, y **outset**. El valor por defecto es **none**, lo que significa que el borde no se mostrará a menos que asignemos un valor diferente a esta propiedad. También podemos declarar los estilos de forma independiente con las propiedades **border-top-style**, **border-bottom-style**, **border-left-style**, y **border-right-style**.

border-color—Esta propiedad define el color del borde. Acepta hasta cuatro valores separados por un espacio para especificar el color de cada lado del borde (superior, derecho, inferior, e izquierdo, en ese orden). También podemos declarar los colores de forma independiente con las propiedades **border-top-color**, **border-bottom-color**, **border-left-color**, y **border-right-color**.

border—Esta propiedad nos permite declarar todos los atributos del borde al mismo tiempo. También podemos usar las propiedades **`border-top`**, **`border-bottom`**, **`border-left`**, y **`border-right`** para definir los valores de cada borde de forma independiente.

Para asignar un borde a un elemento, todo lo que tenemos que hacer es definir el estilo con la propiedad **`border-style`**. Una vez que se define el estilo, el navegador usa los valores por defecto para generar el borde. Si no queremos dejar que el navegador determine estos valores, podemos usar el resto de las propiedades para configurar todos los atributos del borde. El siguiente ejemplo asigna un borde sólido de 2 píxeles de ancho a la cabecera de nuestro documento.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border-style: solid;
  border-width: 2px;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-51: *Agregando un borde a un elemento*

Hojas de Estilo en Cascada

Figura 3-25: *Borde sólido*

Como hemos hecho con la propiedad **`background`**, declaramos todos los valores juntos usando la propiedad **`border`**. El siguiente ejemplo crea un borde discontinuo de 5 píxeles en color gris.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 5px dashed #CCCCCC;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-52: *Definiendo un borde con la propiedad* `border`

Figura 3-26: *Borde discontinuo*

El borde agregado con estas propiedades se dibuja alrededor de la caja del elemento, lo que significa que va a describir un rectángulo con esquinas rectas. Si nuestro diseño requiere esquinas redondeadas, podemos agregar la siguiente propiedad.

border-radius—Esta propiedad define el radio del círculo virtual que el navegador utilizará para dibujar las esquinas redondeadas. Acepta hasta cuatro valores para definir los radios de cada esquina (superior izquierdo, superior derecho, inferior derecho e inferior izquierdo, en ese orden). También podemos usar las propiedades **`border-top-left-radius`**, **`border-top-right-radius`**, **`border-bottom-right-radius`**, y **`border-bottom-left-radius`** para definir el radio de cada esquina de forma independiente.

El siguiente ejemplo genera esquinas redondeadas para nuestra cabecera con un radio de 20 píxeles.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 2px solid;
  border-radius: 20px;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-53: *Generando esquinas redondeadas*

Figura 3-27: *Esquinas redondeadas*

Si todas las esquinas son iguales, podemos declarar solo un valor para esta propiedad, tal como lo hemos hecho en el ejemplo anterior. Sin embargo, al igual que con las propiedades **`margin`** y **`padding`**, si queremos que las esquinas sean diferentes, tenemos que especificar valores diferentes para cada una de ellas.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 2px solid;
  border-radius: 20px 10px 30px 50px;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-54: *Declarando diferentes valores para cada esquina*

En el Listado 3-54, los valores asignados a la propiedad **`border-radius`** representan cuatro ubicaciones diferentes. Los valores se declaran siempre en la dirección de las agujas del reloj, comenzando por la esquina superior izquierda. El orden es: esquina superior izquierda, esquina superior derecha, esquina inferior derecha y esquina inferior izquierda.

Hojas de Estilo en Cascada

Figura 3-28: *Diferentes esquinas*

Lo básico: al igual que las propiedades **`margin`** y **`padding`**, la propiedad **`border-radius`** también puede aceptar solo dos valores. El primer valor se asigna a las esquinas primera y tercera (superior izquierdo e inferior derecho) y a las esquinas segunda y cuarta (superior derecho e inferior izquierdo).

También podemos cambiar la forma de las esquinas agregando valores separados por una barra oblicua. Los valores de la izquierda representan el radio horizontal y los valores de la derecha representan el radio vertical. La combinación de estos valores genera una elipse.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 2px solid;
  border-radius: 20px / 10px;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-55: *Generando esquinas elípticas*

Hojas de Estilo en Cascada

Figura 3-29: *Esquinas elípticas*

Hágalo usted mismo: copie los estilos que desea probar dentro de su archivo CSS y abra el documento en su navegador. Modifique los valores de cada propiedad pare entender cómo trabajan.

Los bordes que acabamos de crear se dibujan en los límites de la caja del elemento, pero también podemos demarcar el elemento con un segundo borde que se dibuja alejado de esto límites. El propósito de estos tipos de bordes es resaltar el elemento. Algunos navegadores lo usan para demarcar texto, pero la mayoría dibuja un segundo borde fuera de los límites de la caja. CSS ofrece las siguientes propiedades para crear este segundo borde.

outline-width—Esta propiedad define el ancho del borde. Acepta valores en cualquiera de las unidades disponibles en CSS (`px`, `%`, `em`, etc.) y también las palabras clave `thin`, `medium`, y `thick`.

outline-style—Esta propiedad define el estilo del borde. Los valores disponibles son `none`, `auto`, `dotted`, `dashed`, `solid`, `double`, `groove`, `ridge`, `inset`, y `outset`.

outline-color—Esta propiedad define el color del borde.

outline-offset—Esta propiedad define el desplazamiento (a qué distancia de los límites de la caja se dibujará el segundo borde). Acepta valores en cualquiera de las unidades disponibles en CSS (`px`, `%`, `em`, etc.).

outline—Esta propiedad nos permite especificar el ancho, estilo y color del borde al mismo tiempo (el desplazamiento aún se debe definir con la propiedad `outline-offset`).

Por defecto, el desplazamiento se declara con el valor 0, por lo que el segundo borde se dibujará a continuación del borde de la caja. Si queremos separar los dos bordes, tenemos que definir la propiedad `outline-offset`.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  outline: 2px dashed #000000;
  outline-offset: 15px;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-56: *Agregando un segundo borde a la cabecera*

En el Listado 3-56, agregamos un segundo borde de 2 píxeles con un desplazamiento de 15 píxeles a los estilos originales asignados a la caja de la cabecera de nuestro documento. El resultado se muestra en la Figura 3-30.

Figura 3-30: *Segundo borde*

Los efectos logrados por las propiedades `border` y `outline` se limitan a simples líneas y unas pocas opciones de configuración, pero CSS nos permite definir bordes personalizados usando imágenes para superar estas limitaciones. Las siguientes son las propiedades que se incluyen con este propósito.

border-image-source—Esta propiedad determina la imagen que se usará para crear el borde. La URL del archivo se declara con la función `url()` (por ejemplo, `url("ladrillos.jpg")`).

border-image-width—Esta propiedad define el ancho del borde. Acepta valores en cualquiera de las unidades disponibles en CSS (`px`, `%`, `em`, etc.).

border-image-repeat—Esta propiedad define cómo se usa la imagen para generar el borde. Los valores disponibles son `repeat`, `round`, `stretch`, y `space`.

border-image-slice—Esta propiedad define cómo se va a cortar la imagen para representar las esquinas del borde. Debemos asignar cuatro valores para especificar los cuatro trozos de la imagen que se utilizarán (si solo se declara un valor, se usa para todos los lados). El valor se puede especificar como un entero o en porcentaje.

border-image-outset—Esta propiedad define el desplazamiento del borde (la distancia a la que se encuentra de la caja del elemento). Acepta valores en cualquiera de las unidades disponibles en CSS (`px`, `%`, `em`, etc.).

border-image—Esta propiedad nos permite especificar todos los atributos del borde al mismo tiempo.

Estas propiedades usan una imagen como patrón. De acuerdo a los valores facilitados, la imagen se corta como un pastel para obtener las piezas necesarias y luego estas piezas se acomodan alrededor del elemento para construir el borde.

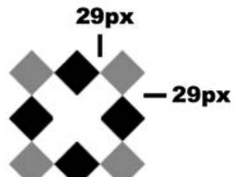

Figura 3-31: *Patrón para construir el borde*

Para lograr este objetivo, necesitamos declarar tres atributos: la ubicación del archivo de la imagen, el tamaño de las piezas que queremos extraer del patrón y las palabras clave que determinan cómo se van a distribuir estas piezas alrededor del elemento.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 29px solid;
  border-image-source: url("diamantes.png");
  border-image-slice: 29;
  border-image-repeat: stretch;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-57: *Creando un borde personalizado para la caja de la cabecera*

En el Listado 3-57, creamos un borde de 29 píxeles para la caja de la cabecera y luego cargamos la imagen diamantes.png para construir el borde. El valor 29 asignado a la propiedad **border-image-slice** declara el tamaño de las piezas, y el valor **stretch**, asignado a la propiedad **border-image-repeat,** es uno de los métodos disponibles para distribuir estas piezas alrededor de la caja. Hay tres valores disponibles para este último atributo. El valor **repeat** repite las piezas tomadas de la imagen cuantas veces sea necesario para cubrir el lado del elemento. En este caso, el tamaño de la pieza se preserva y la imagen solo se corta si no hay espacio suficiente para ubicarla. El valor **round** calcula la longitud del lado de la caja y luego estira las piezas para asegurarse de que no se corta ninguna. Finalmente, el valor **stretch** (usado en el Listado 3-57) estira una pieza hasta cubrir todo el lado.

Figura 3-32: *Borde de tipo* `stretch`

Como siempre, podemos declarar todos los valores al mismo tiempo usando una sola propiedad.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 29px solid;
  border-image: url("diamantes.png") 29 round;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-58: *Definiendo el borde con la propiedad* `border-image`

Figura 3-33: *Borde de tipo* `round`

Sombras

Otro efecto interesante que podemos aplicar a un elemento son las sombras. CSS incluye las siguientes propiedades para generar sombras para la caja de un elemento y también para formas irregulares como texto.

box-shadow—Esta propiedad genera una sombra desde la caja del elemento. Acepta hasta seis valores. Podemos declarar el desplazamiento horizontal y vertical de la sombra, el radio de difuminado, el valor de propagación, el color de la sombra, y también podemos incluir el valor **`inset`** para indicar que la sombra deberá proyectarse dentro de la caja.

text-shadow—Esta propiedad genera una sombra desde un texto. Acepta hasta cuatro valores. Podemos declarar el desplazamiento horizontal y vertical, el radio de difuminado y el color de la sombra.

La propiedad **`box-shadow`** necesita al menos tres valores para poder determinar el color y el desplazamiento de la sombra. El desplazamiento puede ser positivo o negativo. Los valores indican la distancia horizontal y vertical desde la sombra al elemento: los valores negativos posicionan la sombra a la izquierda y encima del elemento, mientras que los positivos crean una sombra a la derecha y debajo del elemento. El valor 0 ubica a la sombra detrás del elemento y ofrece la posibilidad de generar un efecto de difuminado a su alrededor. El siguiente ejemplo agrega una sombra básica a la cabecera de nuestro documento.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  box-shadow: rgb(150,150,150) 5px 5px;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-59: *Agregando una sombra a la cabecera*

Figura 3-34: *Sombra básica*

La sombra que hemos obtenido hasta el momento es sólida, sin gradientes ni transparencia, pero aún no se parece a una sombre real. Para mejorar su aspecto podemos agregar una distancia de difuminado.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  box-shadow: rgb(150,150,150) 5px 5px 20px;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-60: *Agregando el valor de difuminado con la propiedad* `box-shadow`

Hojas de Estilo en Cascada

Figura 3-35: Sombra

Al agregar otro valor en píxeles al final de la propiedad, podemos propagar la sombra. Este efecto cambia levemente la apariencia de la sombra y expande el área que ocupa.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  box-shadow: rgb(150,150,150) 10px 10px 20px 10px;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-61: *Expandiendo la sombra*

Hojas de Estilo en Cascada

Figura 3-36: *Sombra más amplia*

El último valor disponible para la propiedad **box-shadow** no es un número, sino la palabra clave **inset**. Este valor transforma la sombra externa en una sombra interna, lo cual otorga un efecto de profundidad al elemento.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  box-shadow: rgb(150,150,150) 5px 5px 10px inset;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-62: *Creando una sombra interna*

Hojas de Estilo en Cascada

Figura 3-37: *Sombra interna*

IMPORTANTE: las sombras no expanden el elemento ni tampoco incrementan su tamaño, por lo que deberá asegurarse de que haya suficiente espacio alrededor del elemento para que se vea la sombra.

La propiedad **box-shadow** se ha diseñado específicamente para cajas de elementos. Si intentamos aplicar este efecto a un elemento **<span>**, por ejemplo, la sombra se asignará a la caja alrededor del elemento, no a su contenido. CSS define una propiedad aparte para generar la sombra de un texto llamada **text-shadow**.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
}
#titulo {
  font: bold 26px Verdana, sans-serif;
  text-shadow: rgb(150, 150, 150) 3px 3px 5px;
}
```

Listado 3-63: *Agregando una sombra al título*

Los valores de la propiedad **text-shadow** son similares a los asignados a la propiedad **box-shadow**. Podemos especificar el color de la sombra, la distancia horizontal y vertical desde la sombra al elemento, y el radio de difuminado. En el Listado 3-63, se genera una sombra para el título de la cabecera con una distancia de 3 píxeles y un radio de difuminado de 5. El resultado se muestra en la Figura 3-38.

Figura 3-38: *Sombra de texto*

Gradientes

Un gradiente se forma mediante una serie de colores que varían continuamente con una transición suave de un color a otro. Los gradientes se crean como imágenes y se agregan al fondo del elemento con las propiedades **background-image** o **background**. Para crear la imagen con el gradiente, CSS ofrece las siguientes funciones:

linear-gradient(posición, ángulo, colores**)**—Esta función crea un gradiente lineal. El atributo **posición** determina el lado o la esquina desde la cual comienza el gradiente y se declara con los valores **top**, **bottom**, **left** y **right**; el atributo **ángulo** define la dirección del gradiente y se puede declarar en las unidades **deg** (grados), **grad** (gradianes), **rad** (radianes), o **turn** (espiras), y el atributo **colores** es la lista de colores que participan en el gradiente separados por coma. Los valores para el atributo **colores** pueden incluir un segundo valor en porcentaje separado por un espacio para indicar la posición donde finaliza el color.

radial-gradient(posición, forma, colores, extensión**)**—Esta función crea un gradiente radial. El atributo **posición** indica el origen del gradiente y se puede declarar en píxeles, en porcentaje, o por medio de la combinación de los valores `center`, `top`, `bottom`, `left` y `right`, el atributo **forma** determina la forma del gradiente y se declara con los valores `circle` y `ellipse`, el atributo **colores** es la lista de los colores que participan en el gradiente separados por coma, y el atributo **extensión** determina la forma que el gradiente va a adquirir con los valores `closest-side`, `closest-corner`, `farthest-side`, y `farthest-corner`. Los valores para el atributo **colores** pueden incluir un segundo valor en porcentaje separado por un espacio para indicar la posición donde finaliza el color.

Los gradientes se declaran como imágenes de fondo, por lo que podemos aplicarlos a un elemento por medio de las propiedades `background-image` o `background`, como lo hacemos en el siguiente ejemplo.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  background: -webkit-linear-gradient(top, #FFFFFF, #666666);
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-64: *Agregando un gradiente lineal a la cabecera*

Figura 3-39: *Gradiente lineal*

IMPORTANTE: algunas propiedades y funciones CSS todavía se consideran experimentales. Por esta razón, se deben declarar con un prefijo que representa el motor web utilizado. Por ejemplo, si queremos que la función `linear-gradient()` funcione en Google Chrome, tenemos que declararla como `-webkit-linear-gradient()`. Si quiere usar esta función en su sitio web, deberá repetir la propiedad `background` para cada navegador existente con su correspondiente prefijo. Los prefijos requeridos para los navegadores más populares son `-moz-` para Mozilla Firefox, `-webkit-` para Safari y Google Chrome, `-o-` para Opera, y `-ms-` para Internet Explorer.

En el ejemplo anterior, usamos el valor `top` para determinar la posición inicial del gradiente, pero también podemos combinar dos valores para comenzar el gradiente desde una esquina del elemento, como en el siguiente ejemplo.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  background: -webkit-linear-gradient(top right, #FFFFFF, #666666);
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-65: *Estableciendo la posición inicial*

Hojas de Estilo en Cascada

Figura 3-40: *Diferente comienzo para un gradiente lineal*

Cuando trabajamos con gradientes lineales, también podemos configurar la dirección con un ángulo en grados.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  background: -webkit-linear-gradient(30deg, #FFFFFF, #666666);
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-66: *Creando un gradiente con una dirección de 30 grados*

Hojas de Estilo en Cascada

Figura 3-41: *Gradiente lineal con la dirección establecida en grados*

Los gradientes anteriores se han creado con solo dos colores, pero podemos agregar más valores para generar un gradiente multicolor.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  background: -webkit-linear-gradient(top, #000000, #FFFFFF, #999999);
```

```
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-67: *Creando un gradiente multicolor*

Figura 3-42: *Gradiente lineal multicolor*

Si usamos el valor **transparent** en lugar de un color, podemos hacer que el gradiente sea traslúcido y de esta manera se mezcle con el fondo (este efecto también se puede lograr con la función **rgba()** estudiada anteriormente). En el siguiente ejemplo, asignamos una imagen de fondo al elemento **<body>** para cambiar el fondo de la página y poder comprobar que logramos un gradiente translúcido.

```
body {
  background: url("ladrillosclaros.jpg");
}
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  background: -webkit-linear-gradient(top, transparent, #666666);
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-68: *Creando un gradiente translúcido*

Figura 3-43: *Gradiente translúcido*

Los parámetros que definen los colores también pueden determinar el punto de comienzo y final de cada color incluyendo un valor adicional en porcentaje.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  background: -webkit-linear-gradient(top, #FFFFFF 50%, #666666 90%);
}
```

```
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-69: *Configurando los puntos de comienzo y final de cada color*

Hojas de Estilo en Cascada

Figura 3-44: *Gradiente lineal con valores de comienzo y final*

Además de gradientes lineales, también podemos crear gradientes radiales. La sintaxis para gradientes radiales no difiere mucho de los gradientes lineales que acabamos de estudiar. La única diferencia es que tenemos que usar la función **`radial-gradient()`** en lugar de la función **`linear-gradient()`** e incluir un parámetro que determina la forma del gradiente con los valores **`circle`** o **`ellipse`**.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  background: -webkit-radial-gradient(center, ellipse, #FFFFFF,
#000000);
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-70: *Creando un gradiente radial*

Hojas de Estilo en Cascada

Figura 3-45: *Gradiente radial*

Excepto por la forma (círculo o elipse), el resto de esta función trabaja del mismo modo que **`linear-gradient()`**. La posición del gradiente se puede personalizar y podemos usar varios colores con un segundo valor para determinar los límites de cada uno de ellos.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  background: -webkit-radial-gradient(30px 50px, ellipse, #FFFFFF 50%,
#666666 70%, #999999 90%);
}
```

```
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-71: *Creando un gradiente radial multicolor*

Figura 3-46: *Gradiente radial con puntos de inicio y final*

Filtros

Los filtros agregan efectos a un elemento y su contenido. CSS incluye la propiedad `filter` para asignar un filtro a un elemento y las siguiente funciones para crearlo.

blur(valor**)**—Esta función produce un efecto de difuminado. Acepta valores en píxeles desde `1px` a `10px`.

grayscale(value**)**—Esta función convierte los colores de la imagen en una escala de grises. Acepta números decimales entre `0.1` y `1`.

drop-shadow(x, y, tamaño, color**)**—Esta función genera una sombra. Los atributos **x** e **y** determinan la distancia entre la sombra y la imagen, el atributo **tamaño** especifica el tamaño de la sombra, y el atributo **color** declara su color.

sepia(valor**)**—Esta función le otorga un tono sepia (ocre) a los colores de la imagen. Acepta números decimales entre `0.1` y `1`.

brightness(valor**)**—Esta función cambia el brillo de la imagen. Acepta números decimales entre `0.1` y `10`.

contrast(valor**)**—Esta función cambia el contraste de la imagen. Acepta números decimales entre `0.1` y `10`.

hue-rotate(valor**)**—Esta función aplica una rotación a los matices de la imagen. Acepta un valor en grados desde `1deg` a `360deg`.

invert(valor**)**—Esta función invierte los colores de la imagen y produce un negativo. Acepta números decimales entre `0.1` y `1`.

saturate(valor**)**—Esta función satura los colores de la imagen. Acepta números decimales entre `0.1` y `10`.

opacity(valor**)**—Esta función cambia la opacidad de la imagen. Acepta números decimales entre `0` y `1` (`0` es totalmente transparente y `1` totalmente opaco).

Estos filtros no solo se pueden aplicar a imágenes, sino también a otros elementos en el documento. El siguiente ejemplo aplica un efecto de difuminado a la cabecera de nuestro documento.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  filter: blur(5px);
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-72: *Aplicando un filtro a la cabecera*

Figura 3-47: *Una cabecera borrosa*

Hágalo usted mismo: reemplace las reglas en su archivo CSS por las reglas del Listado 3-72 y abra el documento en su navegador. Reemplace la función **`blur()`** con cualquiera de las restantes funciones disponibles para ver cómo trabajan.

Transformaciones

Una vez que se crean los elementos HTML, estos permanecen inmóviles, pero podemos modificar su posición, tamaño y apariencia por medio de la propiedad **`transform`**. Esta propiedad realiza cuatro transformaciones básicas a un elemento: escalado, rotación, inclinación y traslación. Las siguientes son las funciones definidas para este propósito.

scale(x, y**)**—Esta función modifica la escala del elemento. Existen otras dos funciones relacionadas llamadas **`scaleX()`** y **`scaleY()`** pare especificar los valores horizontales y verticales independientemente.

rotate(ángulo**)**—Esta función rota el elemento. El atributo representa los grados de rotación y se puede declarar en **`deg`** (grados), **`grad`** (gradianes), **`rad`** (radianes) o **`turn`** (espiras).

skew(ángulo**)**—Esta función inclina el elemento. El atributo representa los grados de desplazamiento. La función puede incluir dos valores para representar el ángulo horizontal y vertical. Los valores se pueden declarar en **`deg`** (grados), **`grad`** (gradianes), **`rad`** (radianes) o **`turn`** (espiras).

translate(x, y**)**—Esta función desplaza al elemento a la posición determinada por los atributos **x** e **y**.

La función **`scale()`** recibe dos parámetros, el valor **x** para la escala horizontal y el valor **y** para la escala vertical. Si solo se declara un valor, ese mismo valor se usa para ambos parámetros.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  transform: scale(2);
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-73: *Escalando la cabecera*

En el ejemplo del Listado 3-73, transformamos la cabecera con una escala que duplica su tamaño. A la escala se le pueden asignar números enteros y decimales, y esta escala se calcula por medio de una matriz. Los valores entre 0 y 1 reducen el tamaño del elemento, el valor 1 preserva las proporciones originales, mientras que los valores sobre 1 incrementan las dimensiones del elemento de forma lineal.

Cuando asignamos valores negativos a esta función, se genera un efecto interesante.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  transform: scale(1, -1);
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-74: *Creando una imagen espejo con la función* `scale()`

En el Listado 3-74, se declaran dos parámetros para modificar la escala de la cabecera. Ambos valores preservan las proporciones originales, pero el segundo valor es negativo y, por lo tanto, invierte el elemento en el eje vertical, produciendo una imagen invertida.

Figura 3-48: *Imagen espejo con* `scale()`

Además de escalar el elemento, también podemos rotarlo con la función **`rotate()`**. En este caso, los valores negativos cambian la dirección en la cual se rota el elemento. El siguiente ejemplo rota la cabecera 30 grados en el sentido de las agujas del reloj.

```
header {
  margin: 30px;
```

```
  padding: 15px;
  text-align: center;
  border: 1px solid;
  transform: rotate(30deg);
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-75: *Rotando la cabecera*

Otra función disponible para la propiedad **`transform`** es **`skew()`**. Esta función cambia la simetría del elemento en grados y en una o ambas dimensiones.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  transform: skew(20deg);
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-76: *Inclinando la cabecera*

Esta función recibe dos valores, pero a diferencia de otras funciones, cada parámetro solo afecta a una dimensión (los parámetros son independientes). En el Listado 3-76, solo el primer parámetro se declara y, por lo tanto, solo se modifica la dimensión horizontal. Si lo deseamos, podemos usar las funciones adicionales **`skewX()`** y **`skewY()`** para lograr el mismo efecto.

Figura 3-49: *Inclinación horizontal*

La pantalla de un dispositivo se divide en filas y columnas de píxeles (la mínima unidad visual de la pantalla). Con el propósito de identificar la posición de cada píxel, los ordenadores usan un sistema de coordenadas, donde las filas y columnas de píxeles se cuentan desde la esquina superior izquierda a la esquina inferior derecha de la cuadrícula, comenzando por el valor 0 (los valores se incrementan de izquierda a derecha y de arriba abajo). Por ejemplo, el primer píxel de la esquina superior izquierda de la pantalla se encuentra en la posición 0,0 (columna 0, fila 0), mientras que un píxel que se encuentra 30 píxeles del lado izquierdo de la pantalla y 10 píxeles de la parte superior estará ubicado en la posición 30,10.

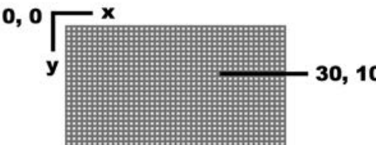

Figura 3-50: *Sistema de coordenadas*

El punto en la posición 0,0 se llama *origen*, y las líneas de columnas y filas se denominan *ejes* y se identifican con las letras **x** e **y** (**x** para columnas e **y** para filas), tal como ilustra la Figura 3-50. Usando la propiedad `transform` podemos cambiar la posición de un elemento en esta cuadrícula. La función que tenemos que asignar a la propiedad en este caso es `translate()`.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  transform: translate(100px);
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-77: *Moviendo la cabecera hacia la derecha*

La función `translate()` utiliza el sistema de coordenadas para establecer la posición del elemento, usando su posición actual como referencia. La esquina superior izquierda del elemento se considera que está en la posición `0,0`, por lo que los valores negativos mueven el elemento hacia la izquierda o encima de la posición original, y los valores positivos lo mueven a la derecha y abajo.

En el Listado 3-77, movemos la cabecera a la derecha 100 píxeles de su posición original. Se pueden declarar dos valores en esta función para mover el elemento horizontalmente y verticalmente, o podemos usar las funciones `translateX()` y `translateY()` para declarar los valores de forma independiente.

A veces, puede resultar útil aplicar varias transformaciones a la vez a un mismo elemento. Para combinar transformaciones con la propiedad `transform`, tenemos que declarar las funciones separadas por un espacio.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  transform: translateY(100px) rotate(45deg) scaleX(0.3);
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-78: *Moviendo, escalando y rotando el elemento con una sola propiedad*

IMPORTANTE: cuando combinamos múltiples funciones, debemos considerar que el orden de combinación es importante. Esto se debe a que algunas funciones modifican el punto de origen y el centro del elemento y, por lo tanto, cambian los parámetros sobre los que el resto de las funciones trabajan.

De la misma manera que podemos generar transformaciones en dos dimensiones sobre elementos HTML, también podemos hacerlo en tres dimensiones. Estos tipos de transformaciones se realizan considerando un tercer eje que representa profundidad y se identifica con la letra **z**. Las siguientes son las funciones disponibles para este propósito.

scale3d(x, y, z**)**—Esta función asigna una nueva escala al elemento en un espacio 3D. Acepta tres valores en números decimales para establecer la escala en los ejes **x**, **y** y **z**. Al igual que con transformaciones 2D, el valor 1 preserva la escala original.

rotate3d(x, y, z, ángulo**)**—Esta función rota el elemento en un ángulo y sobre un eje específicos. Los valores para los ejes se deben especificar en números decimales y el ángulo se puede expresar en **deg** (grados), **grad** (gradianes), **rad** (radianes), o **turn** (espiras). Los valores asignados a los ejes determinan un vector de rotación, por lo que los valores no son importantes, pero sí lo es la relación entre los mismos. Por ejemplo, **rotate3d(5, 2, 6, 30deg)** producirá el mismo efecto que **rotate3d(50, 20, 60, 30deg)**, debido a que el vector resultante es el mismo.

translate3d(x, y, z**)**—Esta función mueve el elemento a una nueva posición en el espacio 3D. Acepta tres valores en píxeles para los ejes **x**, **y** y **z**.

perspective(valor**)**—Esta función agrega un efecto de profundidad a la escena incrementando el tamaño del lado del elemento cercano al espectador.

Algunas transformaciones 3D se pueden aplicar directamente al elemento, como hemos hecho con las transformaciones 2D, pero otras requieren que primero declaremos la perspectiva. Por ejemplo, si rotamos el elemento en el eje **y**, un lado del elemento se desplazará hacia adelante y el otro hacia atrás, pero el tamaño de cada lado permanecerá igual y por ello el usuario no verá la transformación. Para lograr un efecto realista, tenemos que declarar la perspectiva con la función **perspective()**.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;

  border: 1px solid;
  transform: perspective(500px) rotate3d(0, 1, 0, 45deg);
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-79: *Aplicando un efecto tridimensional a la cabecera*

La regla del Listado 3-79 primero asigna el valor de la perspectiva y luego rota el elemento 45 grados en el eje **y** (para seleccionar un eje, tenemos que declarar al resto de los ejes con el valor

0). El navegador rota el elemento y, debido a que definimos la perspectiva para la transformación, expande un lado del elemento y reduce el otro para crear la impresión de perspectiva.

Figura 3-51: *Efecto 3D con perspectiva*

CSS también incluye algunas propiedades que podemos usar para lograr un efecto más realista.

perspective—Esta propiedad trabaja de forma similar a la función **`perspective()`**, pero opera en el elemento padre. La propiedad crea un contenedor que aplica el efecto de perspectiva a los elementos en su interior.

perspective-origin—Esta propiedad cambia las coordenadas **`x`** e **`y`** del espectador. Acepta dos valores en porcentaje, píxeles, o las palabras clave **`center`**, **`left`**, **`right`**, **`top`** y **`bottom`**. Los valores por defecto son **`50% 50%`**.

backface-visibility—Esta propiedad determina si el reverso del elemento será visible o no. Acepta dos valores: **`visible`** o **`hidden`**, con el valor **`visible`** configurado por defecto.

Debido a que en nuestro ejemplo la cabecera del documento es hija directa del cuerpo, tenemos que asignar estas propiedades al elemento **`<body>`**. El siguiente ejemplo define la perspectiva del cuerpo y luego rota la cabecera con la propiedad **`transform`**, como hemos hecho anteriormente.

```
body {
  perspective: 500px;
  perspective-origin: 50% 90%;
}
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  transform: rotate3d(0, 1, 0, 135deg);
}

#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-80: *Declarando un origen diferente para el espectador*

El resultado de aplicar la perspectiva al elemento padre es el mismo que si usáramos la función **`perspective()`** directamente en el elemento que queremos modificar, pero declarando la perspectiva de esta manera podemos usar la propiedad **`perspective-origin`** y al mismo tiempo cambiar las coordenadas del espectador.

Figura 3-52: *Efecto 3D usando la propiedad* `perspective`

Hágalo usted mismo: reemplace las reglas en su archivo CSS por las reglas del Listado 3-80 y abra el documento en su navegador. Agregue la propiedad **`backface-visibility`** con el valor **`hidden`** a la cabecera para volverla invisible cuando está invertida.

Todas las funciones que hemos estudiado hasta el momento modifican los elementos en el documento, pero una vez que la página web se muestra, permanece igual. Sin embargo, podemos combinar transformaciones y seudoclases para convertir nuestro documento en una aplicación dinámica. La seudoclase que podemos implementar en este caso se llama **`:hover`**. Las seudoclases, como hemos visto anteriormente, agregan efectos especiales a las reglas. En este caso, la regla con la seudoclase **`:hover`** se aplica solo cuando el ratón se encuentra sobre el elemento que referencia.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
}
header:hover {
  transform: rotate(5deg);
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-81: *Respondiendo al ratón*

En el Listado 3-81, la regla original de la cabecera es la misma, pero ahora se agrega una nueva regla identificada con el selector **`header:hover`** para aplicar un efecto de transformación cuando el ratón se encuentra sobre el elemento. En consecuencia, cada vez que el usuario mueve el ratón sobre la cabecera, la propiedad **`transform`** rota el elemento 5 grados, y cuando el puntero se mueve fuera de la caja del elemento, el mismo se rota nuevamente a su posición original. Este código logra una animación básica pero útil usando solo propiedades CSS.

Transiciones

Con la seudoclase **`:hover`** podemos realizar transformaciones dinámicas. Sin embargo, una animación real requiere una transición entre los dos pasos del proceso. Para este propósito, CSS ofrece las siguientes propiedades.

transition-property—Esta propiedad especifica las propiedades que participan en la transición. Además de los nombres de las propiedades, podemos asignar el valor **`all`** para indicar que todas las propiedades participarán de la transición.

transition-duration—Esta propiedad especifica la duración de la transición en segundos (**`s`**).

transition-timing-function—Esta propiedad determina la función que se usa para calcular los valores para la transición. Los valores disponibles son **`ease`**, **`ease-in`**, **`ease-out`**, **`ease-in-out`**, **`linear`**, **`step-start`**, y **`step-end`**.

transition-delay—Esta propiedad especifica el tiempo que el navegador esperará antes de iniciar la animación.

transition—Esta propiedad nos permite declarar todos los valores de la transición al mismo tiempo.

Implementando estas propiedades le indicamos al navegador que tiene que crear todos los pasos de la animación y generar una transición entre el estado actual del elemento y el especificado por las propiedades. Las siguientes reglas implementan la propiedad **`transition`** para animar la transformación introducida en el ejemplo anterior.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  transition: transform 1s ease-in-out 0s;
}
header:hover {
  transform: rotate(5deg);
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-82: *Creando una animación con la propiedad* `transition`

La propiedad **`transition`** puede recibir hasta cuatro parámetros separados por un espacio. El primer valor es la propiedad que se considerará para crear la transición (en nuestro ejemplo, usamos la propiedad **`transform`**), el segundo parámetro determina la duración de la animación (1 segundo), el tercer parámetro es un valor que determina cómo se llevará a cabo la transición por medio de una curva Bézier, y el último parámetro determina cuántos segundos tarda la animación en comenzar.

Si la transición tiene que considerar los valores de más de una propiedad, tenemos que declarar los nombres de las propiedades separadas por coma. Cuando se tienen que considerar todas las propiedades que se modifican para crear la animación, podemos usar el valor **`all`** en su lugar.

Hágalo usted mismo: reemplace las reglas en su archivo CSS por las reglas del Listado 3-82 y abra el documento en su navegador. Mueva el puntero del ratón sobre la cabecera para iniciar la animación.

Animaciones

La propiedad `transition` crea una animación básica, pero solo se involucran dos estados en el proceso: el estado inicial determinado por los valores actuales de las propiedades y el estado final, determinado por los nuevos valores. Para crear una animación real, necesitamos declarar más de dos estados, como los fotogramas de una película. CSS incluye las siguientes propiedades para componer animaciones más complejas.

animation-name—Esta propiedad especifica el nombre usado para identificar los pasos de la animación. Se puede usar para configurar varias animaciones al mismo tiempo declarando los nombres separados por coma.

animation-duration—Esta propiedad determina la duración de cada ciclo de la animación. El valor se debe especificar en segundos (por ejemplo, `1s`).

animation-timing-function—Esta propiedad determina cómo se llevará a cabo el proceso de animación por medio de una curva Bézier declarada con los valores `ease`, `linear`, `ease-in`, `ease-out` y `ease-in-out`.

animation-delay—Esta propiedad especifica el tiempo que el navegador esperará antes de iniciar la animación.

animation-iteration-count—Esta propiedad declara la cantidad de veces que se ejecutará la animación. Acepta un número entero o el valor `infinite`, el cual hace que la animación se ejecute por tiempo indefinido. El valor por defecto es `1`.

animation-direction—Esta propiedad declara la dirección de la animación. Acepta cuatro valores: `normal` (por defecto), `reverse`, `alternate`, y `alternate-reverse`. El valor `reverse` invierte la dirección de la animación, mostrando los pasos en la dirección opuesta en la que se han declarado. El valor `alternate` mezcla los ciclos de la animación, reproduciendo los que tienen un índice impar en dirección normal y el resto en dirección invertida. Finalmente, el valor `alternate-reverse` hace lo mismo que `alternate`, pero en sentido inverso.

animation-fill-mode—Esta propiedad define cómo afecta la animación a los estilos del elemento. Acepta los valores `none` (por defecto), `forwards`, `backwards`, y `both`. El valor `forwards` mantiene al elemento con los estilos definidos por las propiedades aplicadas en el último paso de la animación, mientras que `backwards` aplica los estilos del primer paso tan pronto como se define la animación (antes de ser ejecutada). Finalmente, el valor `both` produce ambos efectos.

animation—Esta propiedad nos permite definir todos los valores de la animación al mismo tiempo.

Estas propiedades configuran la animación, pero los pasos se declaran por medio de la regla `@keyframes`. Esta regla se debe identificar con el nombre usado para configurar la animación, y debe incluir la lista de propiedades que queremos modificar en cada paso. La posición de cada paso de la animación se determina con un valor en porcentaje, donde 0 % corresponde al primer fotograma o al comienzo de la animación, y 100 % corresponde al final.

```
header {
  margin: 30px;
  padding: 15px;
```

```
  text-align: center;
  border: 1px solid;
  animation: mianimacion 1s ease-in-out 0s infinite normal none;
}
@keyframes mianimacion {
  0% {
    background: #FFFFFF;
  }
  50% {
    background: #FF0000;
  }
  100% {
    background: #FFFFFF;
  }
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-83: *Creando una animación compleja*

Las reglas del Listado 3-83 crean una animación que cambia los colores del fondo de la cabecera de rojo a blanco. La animación se ha definido mediante la propiedad **animation** con una duración de 1 segundo, y configurado para ejecutarse una y otra vez con el valor **infinite**. La propiedad también asigna el nombre **mianimacion** a la animación para poder configurar luego los pasos con la regla **@keyframes**.

Las propiedades indican en cada paso cómo será afectará al elemento. En este caso, declaramos tres pasos, uno al 0 %, otro al 50 %, y un tercero al 100 %. Cuando se inicia la animación, el navegador asigna al elemento los estilos definidos al 0 % y luego cambia los valores de las propiedades gradualmente hasta llegar a los valores definidos al 50 %. El proceso se repite desde este valor hasta el valor final asignado a la propiedad en el último paso (100 %).

En este ejemplo, definimos el estado inicial de la animación al 0 % y el estado final al 100 %, pero también podemos iniciar la animación en cualquier otro punto y declarar todos los pasos que necesitemos, como en el siguiente ejemplo.

```
header {
  margin: 30px;
  padding: 15px;
  text-align: center;
  border: 1px solid;
  animation: mianimacion 1s ease-in-out 0s infinite normal none;
}
@keyframes mianimacion {
  20% {
    background: #FFFFFF;
  }
  35% {
    transform: scale(0.5);
    background: #FFFF00;
  }
  50% {
    transform: scale(1.5);
```

```
    background: #FF0000;
  }
  65% {
    transform: scale(0.5);
    background: #FFFF00;
  }
  80% {
    background: #FFFFFF;
  }
}
#titulo {
  font: bold 26px Verdana, sans-serif;
}
```

Listado 3-84: *Declarando más pasos para nuestra animación*

En el Listado 3-84, la animación comienza al 20 % y termina al 80 %, e incluye un total de cinco pasos. Cada paso de la animación modifica dos propiedades que incrementan el tamaño del elemento y cambian el color de fondo, excepto el primer paso y el último que solo cambian el color para lograr un efecto de rebote.

Capítulo 4
Diseño web

4.1 Cajas

Como hemos mencionado en el capítulo anterior, los navegadores crean una caja virtual alrededor de cada elemento para determinar el área que ocupan. Para organizar estas cajas en la pantalla, los elementos se clasifican en dos tipos básicos: Block (bloque) e Inline (en línea). La diferencia principal entre estos dos tipos es que los elementos Block tienen un tamaño personalizado y generan saltos de línea, mientras que los elementos Inline tienen un tamaño determinado por su contenido y no generan saltos de línea. Debido a sus características, los elementos Block se colocan de uno en uno en las distintas líneas, y los elementos Inline se colocan uno al lado del otro en la misma línea, a menos que no haya suficiente espacio horizontal disponible, como lo ilustra la Figura 4-1.

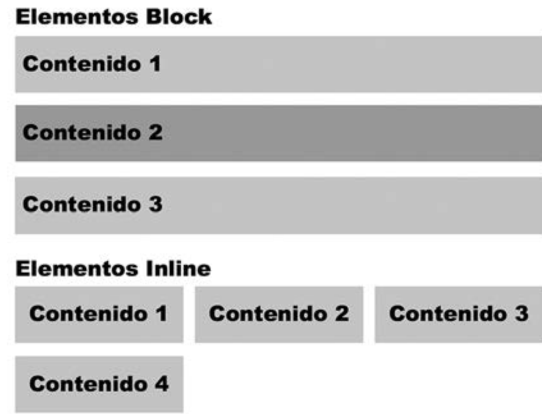

Figura 4-1: Elementos *Block e Inline*

Debido a sus características, los elementos Block son apropiados para crear columnas y secciones en una página web, mientras que los elementos Inline son adecuados para representar contenido. Esta es la razón por la que los elementos que definen la estructura de un documento, como **`<section>`**, **`<nav>`**, **`<header>`**, **`<footer>`**, o **`<div>`**, se declaran como elementos Block por defecto, y otros como **`<span>`**, **`<strong>`**, o **`<em>`**, que representan el contenido de esos elementos, se declaran como elementos Inline.

Display

El que un elemento sea del tipo Block o Inline lo determina el navegador, pero podemos cambiar esta condición desde CSS con la siguiente propiedad.

display—Esta propiedad define el tipo de caja usado para presentar el elemento en pantalla. Existen varios valores disponibles para esta propiedad. Los más utilizados son

none (elimina el elemento), **block** (muestra el elemento en una nueva línea y con un tamaño personalizado), **inline** (muestra el elemento en la misma línea), e **inline-block** (muestra el elemento en la misma línea y con un tamaño personalizado).

Los elementos estructurales se configuran por defecto con el valor **block**, mientras que los elementos que representan el contenido normalmente se configuran como **inline**. Si queremos modificar el tipo de elemento, solo tenemos que asignar la propiedad **display** con un nuevo valor. Así, las antiguas versiones de navegadores no reconocen los nuevos elementos incorporados por HTML5 y los consideran como elementos Inline por defecto. Si queremos asegurarnos de que estos elementos se interpreten como elementos Block en todos los navegadores, podemos declarar la siguiente regla en nuestras hojas de estilo.

```
header, section, main, footer, aside, nav, article, figure, figcaption
{
  display: block;
}
```

Listado 4-1: *Definiendo los elementos HTML5 como elementos Block*

La propiedad **display** cuenta con otros valores además de **block** e **inline**. Por ejemplo, el valor **none** oculta el elemento. Cuando este valor se asigna a un elemento, el documento se presenta como si el elemento no existiera. Es útil cuando queremos cambiar el documento dinámicamente desde JavaScript o cuando usamos diseño web adaptable para diseñar nuestro sitio web, como veremos en próximos capítulos.

Lo básico: el valor **none** para la propiedad **display** elimina el elemento del documento. Si lo que queremos es volver al elemento invisible, podemos usar otra propiedad CSS llamada **visibility**. Esta propiedad acepta los valores **visible** y **hidden**. Cuando el valor **hidden** se asigna a la propiedad, se generan el elemento y su contenido (ocupan un espacio en la pantalla), pero no se muestran al usuario.

Otro valor disponible para la propiedad **display** es **inline-block**. Los elementos Block presentan dos características, una es que producen un salto de línea, por lo que el siguiente elemento se muestra en una nueva línea, y la otra es que pueden adoptar un tamaño personalizado. Esta es la razón por la que las propiedades **width** y **height** estudiadas anteriormente solo trabajan con elementos Block. Si asignamos estas propiedades a un elemento Inline como **<span>**, no ocurre nada. Pero la propiedad **display** ofrece el valor **inline-block** para definir un elemento Inline que puede adoptar un tamaño personalizado. Esto significa que los elementos Inline-Block se posicionarán uno al lado del otro en la misma fila, pero con el tamaño que queramos.

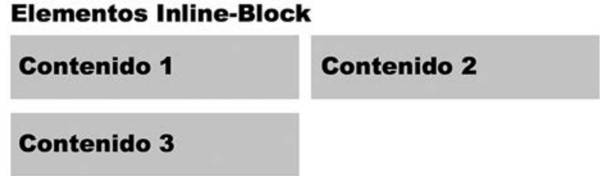

Figura 4-2: *Elementos Inline-Block*

Los elementos Inline-Block nos permiten crear secciones en nuestra página web del tamaño que deseemos y ubicarlas en la misma línea si lo necesitamos. Por ejemplo, si tenemos dos elementos Block que se deben mostrar uno al lado del otro, como los elementos **`<section>`** y **`<aside>`** del documento desarrollado en el Capítulo 2, podemos declararlos como elementos Inline-Block.

Aunque puede resultar tentador usar elementos Inline-Block para diseñar todas las columnas y secciones de nuestras páginas web, CSS incluye mejores propiedades para este propósito. Estas propiedades son parte de lo que llamamos *modelo de cajas*, un conjunto de reglas que determinan cómo se van a mostrar las cajas en pantalla, el espacio que ocupan, y cómo se organizan en la página considerando el espacio disponible.

Actualmente hay varios modelos de cajas disponibles, con el modelo de caja tradicional y el modelo de caja flexible considerados como estándar. Para aprender a diseñar nuestras páginas web debemos entender cómo funcionan estos dos modelos.

4.2 Modelo de caja tradicional

Como ya mencionamos, los elementos Block se colocan unos sobre otros y los elementos Inline se posicionan de izquierda a derecha en la misma línea. El modelo de caja tradicional establece que los elementos pueden flotar a cada lado de la ventana y compartir espacio en la misma línea con otros elementos, sin importar su tipo. Por ejemplo, si tenemos dos elementos Block que representan columnas en el diseño, podemos posicionar una columna a la izquierda y la otra columna a la derecha haciendo que los elementos floten hacia el lado que queremos. Las siguientes son las propiedades que ofrece CSS para este propósito.

float—Esta propiedad permite a un elemento flotar hacia un lado u otro, y ocupar el espacio disponible, incluso cuando tiene que compartir la línea con otro elemento. Los valores disponibles son **`none`** (el elemento no flota), **`left`** (el elemento flota hacia la izquierda) y **`right`** (el elemento flota hacia la derecha).

clear—Esta propiedad restaura el flujo normal del documento, y no permite que el elemento siga flotando hacia la izquierda, la derecha o ambos lados. Los valores disponibles son **`none`**, **`left`**, **`right`**, y **`both`** (ambos).

La propiedad **`float`** hace que el elemento flote a un lado u otro en el espacio disponible. Cuando se aplica esta propiedad, los elementos no siguen el flujo normal del documento, se desplazan a la izquierda o a la derecha del espacio disponible, respondiendo al valor de la propiedad **`float`** y hasta que especifiquemos lo contrario con la propiedad **`clear`**.

Contenido flotante

Las propiedades **`float`** y **`clear`** se usaron originalmente para hacer que el contenido flote alrededor de un elemento. Por ejemplo, si queremos que un texto se muestre al lado de una imagen, podemos hacer flotar la imagen hacia la izquierda o la derecha, y el texto compartirá con la imagen el espacio disponible en la misma línea.

```
<!DOCTYPE html>
<html lang="es">
```

```
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <section>
    <img src="miimagen.jpg" width="120">
    <p>HTML, sigla en inglés de HyperText Markup Language (lenguaje de
marcas de hipertexto), hace referencia al lenguaje de marcado para la
elaboración de páginas web. Se considera el lenguaje web más importante
siendo su invención crucial en la aparición, desarrollo y expansión de
la World Wide Web (WWW).</p>
  </section>
  <footer>
    <p>Publicado por Wikipedia</p>
  </footer>
</body>
</html>
```

Listado 4-2: *Probando la propiedad* `float`

El documento del Listado 4-2 incluye una sección con una imagen y un párrafo. Si abrimos este documento usando los estilos por defecto, el elemento **`<p>`** se muestra debajo del elemento **`<img>`**.

Figura 4-3: *Imagen y párrafo con estilos por defecto*

Si queremos mostrar el texto al lado de la imagen, podemos hacer que el elemento **`<img>`** flote hacia la izquierda o la derecha.

```
section img {
  float: left;
  margin: 0px 10px;
}
```

Listado 4-3: *Flotando la imagen hacia la izquierda*

Cuando un elemento flota hacia un lado, los siguientes elementos flotan a su alrededor ocupando el espacio disponible.

Figura 4-4: *Imagen flota hacia la izquierda*

Si en lugar del valor **`left`** asignamos el valor **`right`** a la propiedad **`float`**, la imagen flota hacia la derecha y el texto la sigue, ocupando el espacio disponible del lado izquierdo.

Figura 4-5: *Imagen flota hacia la derecha*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 4-2 y un archivo CSS con el nombre misestilos.css y la regla del Listado 4-3. Recuerde incluir la imagen miimagen.jpg en el mismo directorio (la imagen está disponible en nuestro sitio web). Abra el documento en su navegador. Debería ver algo similar a lo que muestra la Figura 4-4. Asigne el valor **`right`** a la propiedad **`float`** y actualice el documento en su navegador. La imagen se debería mover hacia la derecha, como muestra la Figura 4-5.

Los navegadores no pueden calcular el tamaño de un contenedor a partir del tamaño de elementos flotantes, por lo que si el elemento afectado por la propiedad **`float`** es más alto que el resto de los elementos de la misma línea, desbordará al contenedor. Por ejemplo, la siguiente figura ilustra lo que ocurre si eliminamos el atributo **`width`** del elemento **`<img>`** en nuestro ejemplo y dejamos que el navegador muestre la imagen en su tamaño original (en este ejemplo, asignamos un fondo gris al elemento **`<section>`** para poder identificar el área que ocupa).

Figura 4-6: *La imagen es más alta que su contenedor*

El navegador estima el tamaño del contenedor de acuerdo al tamaño del texto y, por lo tanto, la imagen se sitúa fuera de los límites del contenedor. Debido a que la imagen flota

hacia la izquierda y se extiende fuera del elemento **<section>**, el contenido del siguiente elemento sigue flotando hasta ocupar el espacio dejado por la imagen y de este modo obtenemos el resultado mostrado en la Figura 4-6 (el elemento **<footer>** se muestra al lado derecho de la imagen en lugar de estar debajo de la misma).

Una forma de asegurarnos de que el contenedor es lo suficientemente alto como para contener los elementos flotantes, es asignándole la propiedad **overflow** con el valor **auto**. Esto fuerza al navegador a calcular el tamaño del contenedor considerando todos los elementos en su interior.

```
section {
  background-color: #CCCCCC;
  overflow: auto;
}
section img {
  float: left;
  margin: 0px 10px;
}
```

Listado 4-4: *Recuperando el flujo normal del documento con la propiedad* overflow

La propiedad **overflow** no deja que el contenido desborde a su contenedor extendiendo el tamaño del contenedor o incluyendo barras de desplazamiento para permitir al usuario ver todo el contenido si el tamaño del contenedor no se puede cambiar.

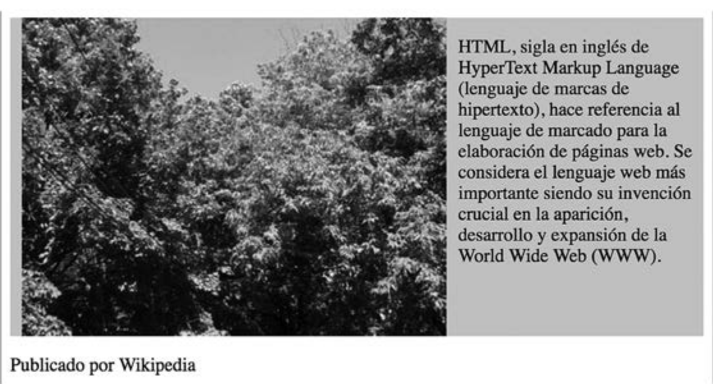

Figura 4-7: *La imagen ya no desborda a su contenedor*

Hágalo usted mismo: elimine el atributo **width** del elemento **<img>** en el documento HTML del Listado 4-2. Reemplace las reglas en su archivo CSS por el código del Listado 4-4. Abra el documento en su navegador. Debería ver algo parecido a la Figura 4-7.

Otra alternativa para normalizar el flujo del documento es declarar la propiedad **clear** en el elemento que se encuentra a continuación del elemento flotante dentro del contenedor. Debido a que no siempre tenemos un elemento hermano después de un elemento flotante que podamos usar para prevenir que los elementos sigan flotando, esta técnica requiere que agreguemos un elemento adicional al documento. Por ejemplo, podemos incluir un elemento

<div> debajo del elemento **<p>** dentro del elemento **<section>** de nuestro documento para impedir que los siguientes elementos continúen flotando hacia los lados.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <section>
    <img src="miimagen.jpg">
    <p>HTML, sigla en inglés de HyperText Markup Language (lenguaje de
marcas de hipertexto), hace referencia al lenguaje de marcado para la
elaboración de páginas web. Se considera el lenguaje web más importante
siendo su invención crucial en la aparición, desarrollo y expansión de
la World Wide Web (WWW).</p>
    <div class="clearfloat"></div>
  </section>
  <footer>
    <p>Publicado por Wikipedia</p>
  </footer>
</body>
</html>
```

***Listado 4-5:** Agregando un elemento vacía para aplicar la propiedad* `clear`

En este caso, debemos asignar la propiedad **clear** al elemento adicional. La propiedad ofrece valores para restaurar el flujo de un lado o ambos. Debido a que normalmente necesitamos restaurar el flujo normal del documento en ambos lados, el valor preferido es **both**.

```
section {
  background-color: #CCCCCC;
}
section img {
  float: left;
  margin: 0px 10px;
}
.clearfloat {
  clear: both;
}
```

***Listado 4-6:** Restaurando el flujo normal del documento con la propiedad* `clear`

Con las reglas del Listado 4-6, el elemento **<div>** restaura el flujo normal del documento, permitiendo al navegador calcular el tamaño del contenedor a partir de su contenido, con lo que obtenemos un resultado similar al que vimos en la Figura 4-7.

Hágalo usted mismo: actualice su archivo HTML con el documento del Listado 4-5 y reemplace las reglas en su archivo CSS por el código del Listado 4-6. Abra el documento en su navegador. Debería ver algo similar a la Figura 4-7.

Cajas flotantes

La propiedad **clear** no afecta a otros aspectos del elemento y no agrega barras de desplazamiento como la propiedad **overflow**. Por lo tanto, es la que se implementa en el modelo de caja tradicional para organizar la estructura de un documento. Con las propiedades **float** y **clear** podemos controlar con precisión dónde se mostrarán los elementos en pantalla y diseñar nuestras páginas web. Por ejemplo, el siguiente documento incluye un elemento **<section>** con cuatro elementos **<div>** que debemos hacer flotar a un lado para convertirlos en columnas dentro de la página, y un elemento **<div>** al final que usaremos para recuperar el flujo normal del documento.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <section id="cajapadre">
    <div id="caja-1">Caja 1</div>
    <div id="caja-2">Caja 2</div>
    <div id="caja-3">Caja 3</div>
    <div id="caja-4">Caja 4</div>
    <div class="restaurar"></div>
  </section>
</body>
</html>
```

Listado 4-7: *Creando un documento para probar la propiedad* float

Como ha ocurrido con otros documentos anteriormente, si abrimos este documento sin asignar estilos personalizados, el contenido de sus elementos se muestra de arriba abajo, siguiendo el flujo del documento por defecto.

Caja 1
Caja 2
Caja 3
Caja 4

Figura 4-8: *Flujo normal del documento*

Debido a que queremos que estas cajas se ubiquen una al lado de la otra representando columnas de nuestra página web, tenemos que asignarles un tamaño fijo y hacerlas flotar a un lado o al otro. El tamaño se determina mediante las propiedades **width** y **height**, y el modo

en el que flotan lo determina la propiedad **float**, pero el valor asignado a esta última propiedad depende de lo que queremos lograr. Si flotamos las cajas hacia la izquierda, estas se alinearán de izquierda a derecha, y si las hacemos flotar hacia la derecha, harán lo mismo pero de derecha a izquierda. Por ejemplo, si las hacemos flotar a la izquierda, **caja-1** se colocará en primer lugar en el lado izquierdo de la caja padre, y luego el resto de las cajas se ubicarán a su derecha en el orden en el que se han declarado en el código (las cajas flotan hacia la izquierda hasta que colisionan con el límite del contenedor o la caja anterior).

```
#cajapadre {
  width: 600px;
  border: 1px solid;
}
#caja-1 {
  float: left;
  width: 140px;
  height: 50px;
  margin: 5px;
  background-color: #AAAAAA;
}
#caja-2 {
  float: left;
  width: 140px;
  height: 50px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-3 {
  float: left;
  width: 140px;
  height: 50px;
  margin: 5px;
  background-color: #AAAAAA;
}
#caja-4 {
  float: left;
  width: 140px;
  height: 50px;
  margin: 5px;
  background-color: #CCCCCC;
}
.restaurar {
  clear: both;
}
```

Listado 4-8: *Flotando cajas a la izquierda*

En el ejemplo del Listado 4-8, asignamos un ancho de 600 píxeles al elemento **<section>** y un tamaño de 140 píxeles por 50 píxeles a cada caja. Siguiendo el flujo normal, estas cajas se apilarían una encima de la otra, pero debido a que les asignamos la propiedad **float** con el valor **left**, flotan hacia la izquierda hasta que se encuentran con el límite del contenedor u otra caja, llenando el espacio disponible en la misma línea.

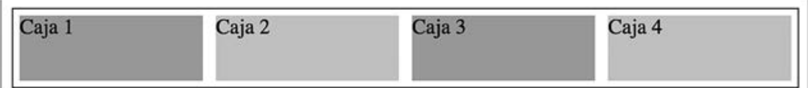

Figura 4-9: *Cajas organizadas con la propiedad* `float`

Como mencionamos anteriormente, cuando el contenido de un elemento flota, el elemento padre no puede calcular su propia altura desde la altura de su contenido. Esta es la razón por la que, cada vez que tenemos elementos que flotan, debemos recuperar el flujo normal en el elemento siguiente con la propiedad **`clear`**.

Hágalo usted mismo: cree un nuevo archivo HTML con el código del Listado 4-7 y un nuevo archivo CSS llamado misestilos.css con las reglas del Listado 4-8. Abra el documento en su navegador. Debería ver algo similar a la Figura 4-9.

En el último ejemplo, nos aseguramos de que el elemento padre es suficientemente ancho como para contener todas las cajas y, por lo tanto, todas se muestran en la misma línea, pero si el contenedor no tiene espacio suficiente, los elementos que no se pueden ubicar en la misma línea se moverán a una nueva. La siguiente regla reduce el tamaño del elemento **`<section>`** a 500 píxeles.

```
#cajapadre {
  width: 500px;
  border: 1px solid;
}
```

Listado 4-9: *Reduciendo el tamaño del contenedor*

Después de aplicar esta regla a nuestro documento, la última caja no encuentra espacio suficiente al lado derecho de la tercera caja y, por lo tanto, flota hacia el lado izquierdo del contenedor en una nueva línea.

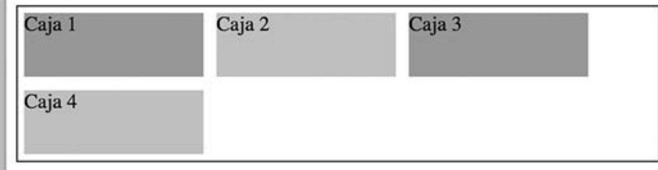

Figura 4-10: *Las cajas llenan el espacio disponible*

Por otro lado, si tenemos más espacio disponible en el contenedor del que necesitan las cajas, el espacio restante se ubica a los lados (izquierdo o derecho, dependiendo del valor de la propiedad **`float`**). Si queremos ubicar el espacio restante en medio de las cajas, podemos hacer flotar algunas cajas hacia la izquierda y otras hacia la derecha. Por ejemplo, las siguientes reglas asignan un tamaño de 120 píxeles a cada caja, dejando un espacio vacío de 80 píxeles, pero como las dos últimas cajas flotan a la derecha en lugar de a la izquierda, el espacio libre se ubica en el medio del contenedor, no a los lados.

```
#cajapadre {
  width: 600px;
  border: 1px solid;
}
#caja-1 {
  float: left;
  width: 120px;
  height: 50px;
  margin: 5px;
  background-color: #AAAAAA;
}
#caja-2 {
  float: left;
  width: 120px;
  height: 50px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-3 {
  float: right;
  width: 120px;
  height: 50px;
  margin: 5px;
  background-color: #AAAAAA;
}
#caja-4 {
  float: right;
  width: 120px;
  height: 50px;
  margin: 5px;
  background-color: #CCCCCC;
}
.restaurar {
  clear: both;
}
```

Listado 4-10: *Flotando las cajas a izquierda y derecha*

Los elementos **caja-1** y **caja-2** flotan a la izquierda, lo que significa que **caja-1** se ubicará en el lado izquierdo del contenedor y **caja-2** se ubicará en el lado derecho de **caja-1**, pero **caja-3** y **caja-4** flotan a la derecha, por lo que van a estar ubicadas en el lado derecho del contenedor, dejando un espacio en el medio.

Figura 4-11: *Espacio libre entre las cajas*

Debido al orden en el que se han declarado los elementos en el código, el elemento **caja-4** se ubica en el lado izquierdo del elemento **caja-3**. El navegador procesa los elementos en el orden en el que se han declarado en el documento; por lo tanto, cuando se procesa el

elemento **caja-3**, se mueve a la derecha hasta que alcanza el límite derecho del contenedor, pero cuando se procesa el elemento **caja-4**, no se puede mover hacia el lado derecho del contenedor porque el elemento **caja-3** ya ocupa ese lugar y, por lo tanto, se ubica en el lado izquierdo de **caja-3**. Si queremos que las cajas se muestren en orden, tenemos que modificar el documento del Listado 4-7 y mover el elemento **<div>** identificado con el nombre **caja-4** sobre el elemento **<div>** identificado con el nombre **caja-3**.

Posicionamiento absoluto

La posición de un elemento puede ser relativa o absoluta. Con una posición es relativa, las cajas se colocan una después de la otra en el espacio designado por el contenedor. Si el espacio no es suficiente o los elementos no son flotantes, las cajas se colocan en una nueva línea. Este es el modo de posicionamiento por defecto, pero existe otro modo llamado *posicionamiento absoluto*. El posicionamiento absoluto nos permite especificar las coordenadas exactas en las que queremos posicionar cada elemento. Las siguientes son las propiedades disponibles para este propósito.

position—Esta propiedad define el tipo de posicionamiento usado para colocar un elemento. Los valores disponibles son **static** (se posiciona de acuerdo con el flujo normal del documento), **relative** (se posiciona según la posición original del elemento), **absolute** (se posiciona con una posición absoluta relativa al contenedor del elemento), y **fixed** (se posiciona con una posición absoluta relativa a la ventana del navegador).

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.

Las propiedades **top**, **bottom**, **left**, y **right** se aplican en ambos tipos de posicionamiento, relativo o absoluto, pero trabajan de diferentes maneras. Cuando el elemento se ubica con posicionamiento relativo, el elemento se desplaza pero el diseño no se modifica. Con posicionamiento absoluto, el elemento se elimina del diseño, por lo que el resto de los elementos también se desplazan para ocupar el nuevo espacio libre.

El siguiente ejemplo usa posicionamiento relativo para desplazar la primera caja de nuestro ejemplo 25 píxeles hacia abajo.

```
#caja-1 {
  position: relative;
  top: 25px;

  float: left;
  width: 140px;
  height: 50px;
  margin: 5px;
```

```
  background-color: #AAAAAA;
}
```

Listado 4-11: *Especificando la posición relativa de un elemento*

Figura 4-12: *Posición relativa*

Hágalo usted mismo: reemplace la regla `caja-1` en su hoja de estilo CSS por la regla del Listado 4-11. En este ejemplo, solo desplazamos la primera caja, pero no tocamos las reglas para el resto de las cajas. Agregue otras propiedades como `left` o `right` para ver cómo afectan a la posición del elemento.

Otra alternativa es usar posicionamiento absoluto. En este caso, el elemento se elimina del diseño, por lo que el resto de los elementos se ven afectados por la regla. Cuando usamos posicionamiento absoluto, también tenemos que considerar que el elemento se ubicará con respecto a la ventana del navegador, a menos que declaremos la posición de su elemento padre. Por lo tanto, si queremos especificar una posición absoluta para un elemento basada en la posición de su elemento padre, también tenemos que declarar la propiedad `position` para el padre. En el siguiente ejemplo, declaramos una posición relativa para el elemento `cajapadre` y una posición absoluta de 25 píxeles desde la parte superior para la `caja-1`, lo que hará que se ubique en una posición más baja que el resto de las cajas.

```
#cajapadre {
  position: relative;
  width: 600px;
  border: 1px solid;
}
#caja-1 {
  position: absolute;
  top: 25px;

  float: left;
  width: 140px;
  height: 50px;
  margin: 5px;
  background-color: #AAAAAA;
}
```

Listado 4-12: *Especificando la posición absoluta de un elemento*

Debido a que el posicionamiento absoluto elimina al elemento del diseño del documento, el resto de las cajas se mueven a la izquierda para ocupar el espacio vacío que ha dejado `caja-1`.

Figura 4-13: *Posición absoluta*

Hágalo usted mismo: reemplace las reglas **#cajapadre** y **#caja-1** en su hoja de estilo CSS con las reglas del Listado 4-12 y abra el documento en su navegador. Debería ver algo parecido a la Figura 4-13.

El orden de los elementos del código no solo determina la ubicación de las cajas en la página, sino también qué caja va a estar por encima de las demás cuando se superponen. Debido a que en nuestro ejemplo **caja-1** se ha declarado primero en el código, se dibuja sobre **caja-2**, pero CSS ofrece la siguiente propiedad para cambiar este comportamiento.

z-index—Esta propiedad define un índice que determina la posición del elemento en el eje **z**. El elemento con el índice más alto se dibujará sobre el elemento con el índice más bajo.

Por ejemplo, podemos mover el elemento **caja-1** debajo del elemento **caja-2** y sobre el elemento **cajapadre** asignando índices negativos a **caja-1** y **cajapadre**.

```
#cajapadre {
  position: relative;
  width: 600px;
  border: 1px solid;
  z-index: -2;
}
#caja-1 {
  position: absolute;
  top: 25px;
  z-index: -1;

  float: left;
  width: 140px;
  height: 50px;
  margin: 5px;
  background-color: #AAAAAA;
}
```

Listado 4-13: *Especificando el índice z*

Los índices negativos se consideran más bajos que los índices asignados por defecto. En el Listado 4-13, asignamos el valor -1 al elemento **caja-1** y el valor -2 al elemento **cajapadre**. Esto mueve **caja-1** debajo de **caja-2,** pero mantiene **caja-1** sobre **cajapadre**, porque su índice es más alto.

Figura 4-14: Índice *z*

Cuando usamos posicionamiento relativo y absoluto, el diseño del documento se modifica y, por lo tanto, esta técnica no se usa para organizar los elementos en pantalla, sino más bien para crear efectos en los cuales los elementos ocultos se muestran respondiendo a acciones del usuario, como cuando necesitamos crear menús desplegables o listas desplazables que revelan información adicional. Por ejemplo, podemos mostrar una caja con el título de una imagen cuando el usuario mueve el ratón sobre ella.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <section id="cajapadre">
    <img src="miimagen.jpg" width="365" height="274">
    <div id="contenedor">
      <div id="contenedorsuperior"></div>
      <div id="contenedorinferior">
        <span><strong>Este es mi patio</strong></span>
      </div>
    </div>
  </section>
</body>
</html>
```

Listado 4-14: *Mostrando una etiqueta desplegable con información adicional*

El documento del Listado 4-14 incluye una sección que contiene una imagen. Debajo del elemento **<img>** se encuentra un elemento **<div>** identificado con el nombre **contenedor** y dentro de este elemento tenemos dos elementos **<div>** más, uno vacío y el otro con un título. El propósito de este código es cubrir la imagen con una caja que, cuando se mueve hacia arriba, revela otra caja en la parte inferior con el título de la imagen. La Figura 4-15 ilustra la estructura generada por estos elementos.

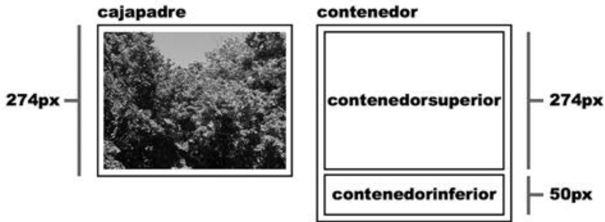

Figura 4-15: *Estructura para presentar una etiqueta desplegable con información adicional*

La imagen y el contenedor se encuentran dentro del elemento **cajapadre**, pero el contenedor tiene que colocarse encima de la imagen, de modo que se pueda mover para revelar el elemento **contenedorinferior** con el título cuando el usuario posiciona el ratón sobre la imagen. Por esta razón, tenemos que asignar una posición absoluta a este elemento.

```
#cajapadre {
  position: relative;
  width: 365px;
  height: 274px;
  overflow: hidden;
}
```

```
#contenedor {
  position: absolute;
  top: 0px;
  width: 365px;
  height: 324px;
}
#contenedorsuperior {
  width: 365px;
  height: 274px;
}
#contenedorinferior {
  width: 365px;
  height: 35px;
  padding-top: 15px;
  background-color: rgba(200, 200, 200, 0.8);
  text-align: center;
}
#contenedor:hover {
  top: -50px;
}
```

Listado 4-15: *Configurando las cajas*

La imagen que usamos en este ejemplo tiene un tamaño de 365 píxeles de ancho por 274 píxeles de alto, por lo que tenemos que especificar este tamaño para el elemento **`cajapadre`**. El contenedor, por otro lado, tiene que ser más alto porque debe contener el elemento **`contenedorsuperior`** y el elemento **`contenedorinferior`** que se revela cuando se mueve hacia arriba. Debido a que la caja con el título tiene una altura de 50 píxeles (35 píxeles de altura y 15 píxeles de relleno), le asignamos a la caja contenedora una altura de 324 píxeles (274 + 50).

La razón por la que ubicamos un contenedor por encima de la imagen es porque tenemos que reaccionar cuando el usuario mueve el ratón sobre la imagen. CSS únicamente nos permite hacerlo con la seudoclase **`:hover`**, como hemos visto anteriormente (vea los Listados 3-81 y 3-82). El problema de esta seudoclase es que solo nos permite modificar el elemento al que se ha aplicado. Usándola con la imagen, solo podríamos modificar la imagen misma, pero aplicándola al contenedor, podemos cambiar el valor de su propiedad **`top`** para moverlo hacia arriba y revelar el elemento **`contenedorinferior`**. La regla al final del Listado 4-15 realiza esta tarea. Cuando el usuario mueve el ratón sobre la imagen, la regla **`#contenedor:hover`** asigna un valor de -50 píxeles a la propiedad **`top`** del elemento **`contenedor`**, moviendo el elemento y su contenido hacia arriba, para revelar el título de la imagen.

Figura 4-16: *Etiqueta sobre la imagen*

El elemento **contenedorinferior** se muestra tan pronto como el ratón se mueve sobre la imagen. Esto se debe a que no declaramos ninguna transición para la propiedad **top**. El valor va de 0px a -50px instantáneamente, por lo que no vemos ninguna transición en el proceso. Para declarar pasos intermedios y crear una animación, tenemos que agregar la propiedad **transition** al elemento **contenedor**.

```
#contenedor {
  position: absolute;
  top: 0px;
  width: 365px;
  height: 324px;
  transition: top 0.5s ease-in-out 0s;
}
```

Listado 4-16: *Animando la etiqueta*

Hágalo usted mismo: cree un nuevo archivo HTML con el código del Listado 4-14 y un archivo CSS llamado misestilos.css con el código del Listado 4-15. Descargue la imagen miimagen.jpg desde nuestro sitio web. Abra el documento en su navegador y mueva el ratón sobre la imagen. Debería ver la etiqueta con el título de la imagen aparecer y desaparecer en la parte inferior de la imagen (Figura 4-16). Reemplace la regla **#contenedor** de su hoja de estilo con la regla del Listado 4-16. Ahora la etiqueta debería ser animada.

Columnas

Además de las propiedades que hemos estudiado para organizar las caja en pantalla, CSS también incluye un grupo de propiedades para facilitar la creación de columnas.

column-count—Esta propiedad especifica el número de columnas que el navegador tiene que generar para distribuir el contenido del elemento.

column-width—Esta propiedad declara el ancho de las columnas. El valor se puede declarar como **auto** (por defecto) o en cualquiera de las unidades de CSS, como píxeles o porcentaje.

column-span—Esta propiedad se aplica a elementos dentro de la caja para determinar si el elemento se ubicará en una columna o repartido entre varias columnas. Los valores disponibles son **all** (todas las columnas) y **none** (por defecto).

column-fill—Esta propiedad determina cómo se repartirá el contenido entre las columnas. Los valores disponibles son **auto** (las columnas son completadas de forma secuencial) y **balance** (el contenido se divide en partes iguales entre todas las columnas).

column-gap—Esta propiedad define el tamaño del espacio entre las columnas. Acepta un valor en cualquiera de las unidades disponibles en CSS, como píxeles y porcentaje.

columns—Esta propiedad nos permite declarar los valores de las propiedades **column-count** y **column-width** al mismo tiempo.

El elemento, cuyo contenido queremos dividir en columnas, es un elemento común, y su contenido se declara del mismo modo que lo hemos hecho antes. El navegador se encarga de

crear las columnas y dividir el contenido por nosotros. El siguiente ejemplo incluye un elemento **<article>** con un texto extenso que vamos a presentar en dos columnas.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <section>
    <article id="articulonoticias">
      <span>HTML, sigla en inglés de HyperText Markup Language
(lenguaje de marcas de hipertexto), hace referencia al lenguaje de
marcado para la elaboración de páginas web. Es un estándar que sirve de
referencia del software que conecta con la elaboración de páginas web
en sus diferentes versiones, define una estructura básica y un código
(denominado código HTML) para la definición de contenido de una página
web, como texto, imágenes, vídeos, juegos, entre otros. Es un estándar
a cargo del World Wide Web Consortium (W3C) o Consorcio WWW,
organización dedicada a la estandarización de casi todas las
tecnologías ligadas a la web, sobre todo en lo referente a su escritura
e interpretación.</span>
    </article>
  </section>
</body>
</html>
```

Listado 4-17: *Dividiendo artículos en columnas*

Las propiedades para generar las columnas se deben aplicar al contenedor. La siguiente regla incluye la propiedad **column-count** para dividir el contenido del elemento **<article>** en dos columnas.

```
#articulonoticias {
  width: 580px;
  padding: 10px;
  border: 1px solid;

  column-count: 2;
  column-gap: 20px;
}
```

Listado 4-18: *Definiendo las columnas*

La regla del Listado 4-18 asigna un tamaño de 580 píxeles al elemento **<article>** e identifica el área ocupada por su contenedor con un borde sólido. La regla también incluye un relleno de 10 píxeles para separar el texto del borde. El resto de las propiedades dividen el contenido en dos columnas con un espacio intermedio de 20 píxeles. El resultado se muestra en la Figura 4-17.

HTML, sigla en inglés de HyperText Markup Language (lenguaje de marcas de hipertexto), hace referencia al lenguaje de marcado para la elaboración de páginas web. Es un estándar que sirve de referencia del software que conecta con la elaboración de páginas web en sus diferentes versiones, define una estructura básica y un código (denominado código HTML) para la definición de contenido de una página web, como texto, imágenes, videos, juegos, entre otros. Es un estándar a cargo del World Wide Web Consortium (W3C) o Consorcio WWW, organización dedicada a la estandarización de casi todas las tecnologías ligadas a la web, sobre todo en lo referente a su escritura e interpretación.

Figura 4-17: *Contenido en dos columnas*

La propiedad `column-gap` define el tamaño del espacio entre las columnas. Esta separación es solo espacio vacío, pero CSS ofrece las siguientes propiedades para generar una línea que ayuda al usuario a visualizar la división.

column-rule-style—Esta propiedad define el estilo de la línea usada para representar la división. Los valores disponibles son **`hidden`** (por defecto), **`dotted`**, **`dashed`**, **`solid`**, **`double`**, **`groove`**, **`ridge`**, **`inset`**, y **`outset`**.

column-rule-color—Esta propiedad especifica el color de la línea usada para representar la división.

column-rule-width—Esta propiedad especifica el ancho de la línea usada para representar la división.

column-rule—Esta propiedad nos permite definir todos los valores de la línea al mismo tiempo.

Para dibujar una línea en medio de las columnas, tenemos que especificar al menos su estilo. El siguiente ejemplo implementa la propiedad `column-rule` para crear una línea negra de 1 píxel.

```
#articulonoticias {
  width: 580px;
  padding: 10px;
  border: 1px solid;

  column-count: 2;
  column-gap: 20px;
  column-rule: 1px solid #000000;
}
```

Listado 4-19: *Agregando una línea entre las columnas*

HTML, sigla en inglés de HyperText Markup Language (lenguaje de marcas de hipertexto), hace referencia al lenguaje de marcado para la elaboración de páginas web. Es un estándar que sirve de referencia del software que conecta con la elaboración de páginas web en sus diferentes versiones, define una estructura básica y un código (denominado código HTML) para la definición de contenido de una página web, como texto, imágenes, videos, juegos, entre otros. Es un estándar a cargo del World Wide Web Consortium (W3C) o Consorcio WWW, organización dedicada a la estandarización de casi todas las tecnologías ligadas a la web, sobre todo en lo referente a su escritura e interpretación.

Figura 4-18: *Columnas separadas por una línea*

Hágalo usted mismo: cree un nuevo archivo HTML con el código del Listado 4-17 y un archivo CSS llamado misestilos.css con el código del Listado 4-18. Abra el documento en su navegador. Debería ver el texto dividido en dos columnas. Reemplace la regla **#articulosnoticias** por la regla del Listado 4-19 y actualice la página en su navegador. Debería ver una línea dividiendo las columnas, tal como muestra la Figura 4-18.

Aplicación de la vida real

El propósito del modelo de caja tradicional es el de organizar la estructura visual de una página web, pero debido a sus características y limitaciones, se deben modificar los documentos para trabajar con este modelo.

Como hemos explicado en el Capítulo 2, los sitios web siguen un patrón estándar y los elementos HTML se han diseñado para crear este diseño, pero no pueden predecir todos los escenarios posibles. Para lograr que los elementos representen las áreas visuales de una página web tradicional, debemos combinarlos con otros elementos. Un truco muy común es envolver los elementos en un elemento contenedor para poder moverlos a las posiciones que deseamos. Por ejemplo, si queremos que la cabecera de nuestro documento sea tan ancha como la ventana del navegador, pero que su contenido se encuentre centrado en la pantalla, podemos envolver el contenido en un elemento **<div>** y luego centrar este elemento en la página. Las siguientes son las adaptaciones que tenemos que hacer al documento introducido en el Capítulo 2 para poder reproducir este tipo de diseño.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <header id="cabeceralogo">
    <div>
      <h1>Este es el título</h1>
    </div>
  </header>
  <nav id="menuprincipal">
    <div>
      <ul>
        <li><a href="index.html">Principal</a></li>
        <li><a href="fotos.html">Fotos</a></li>
        <li><a href="videos.html">Videos</a></li>
        <li><a href="contacto.html">Contacto</a></li>
      </ul>
    </div>
  </nav>
  <main>
    <div>
      <section id="articulosprincipales">
```

```
        <article>
          <h1>Título Primer Artículo</h1>
          <time datetime="2016-12-23" pubdate>
            <div class="numerodia">23</div>
            <div class="nombredia">Viernes</div>
          </time>
          <p>Este es el texto de mi primer artículo</p>
          <figure>
            <img src="miimagen.jpg">
          </figure>
        </article>
        <article>
          <h1>Título Segundo Artículo</h1>
          <time datetime="2016-12-7" pubdate>
            <div class="numerodia">7</div>
            <div class="nombredia">Miércoles</div>
          </time>
          <p>Este es el texto de mi segundo artículo</p>
          <figure>
            <img src="miimagen.jpg">
          </figure>
        </article>
      </section>
      <aside id="infoadicional">
        <h1>Información Personal</h1>
        <p>Cita del artículo uno</p>
        <p>Cita del artículo dos</p>
      </aside>
      <div class="recuperar"></div>
    </div>
  </main>
  <footer id="pielogo">
    <div>
      <section class="seccionpie">
        <h1>Sitio Web</h1>
        <p><a href="index.html">Principal</a></p>
        <p><a href="fotos.html">Fotos</a></p>
        <p><a href="videos.html">Videos</a></p>
      </section>
      <section class="seccionpie">
        <h1>Ayuda</h1>
        <p><a href="contacto.html">Contacto</a></p>
      </section>
      <section class="seccionpie">
        <address>Toronto, Canada</address>
        <small>&copy; Derechos Reservados 2016</small>
      </section>
      <div class="recuperar"></div>
    </div>
  </footer>
</body>
</html>
```

Listado 4-20: *Definiendo un documento para implementar el modelo de caja tradicional*

En este ejemplo, hemos envuelto el contenido de algunos de los elementos estructurales con un elemento **<div>** adicional. Ahora podemos asignar tamaños y alineamientos independientes para cada sección del documento.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 4-20 y un archivo CSS llamado misestilos.css para incluir todos los códigos CSS que vamos a presentar a continuación. Abra el documento en su navegador para ver cómo se organizan los elementos por defecto.

Con el documento HTML listo, es hora de desarrollar nuestra hoja de estilo CSS. A este respecto, lo primero que debemos considerar es qué vamos a hacer con los estilos asignados por defecto por el navegador. En la mayoría de los casos, estos estilos no solo son diferentes de lo que necesitamos, sino que además pueden afectar de forma negativa a nuestro diseño. Por ejemplo, los navegadores asignan márgenes a los elementos que usamos frecuentemente en nuestro documento, como el elemento **<p>**. El elemento **<body>** también genera un margen alrededor de su contenido, lo que hace imposible extender otros elementos hasta los límites de la ventana del navegador. Como si esto fuera poco, la forma en la que se configuran los elementos por defecto difiere de un navegador a otro, especialmente cuando consideramos ediciones de navegadores antiguas que aún se encuentran en uso. Para poder crear un diseño coherente, cualquiera que sea el dispositivo en el que se abre, tenemos que resetear algunos de los estilos por defecto, o todos. Una forma práctica de hacerlo es usando un selector CSS llamado *selector universal*. Se trata de un selector que se representa con el carácter * y que referencia todos los elementos del documento. Por ejemplo, la siguiente regla declara un margen y un relleno de 0 píxeles para todos los elementos de nuestro documento.

```
* {
  margin: 0px;
  padding: 0px;
}
```

Listado 4-21: *Implementando una regla de reseteado*

La primera regla de nuestro archivo CSS introducida en el Listado 4-21 se asegura de que todo elemento tenga un margen y un relleno de 0 píxeles. De ahora en adelante, solo tendremos que modificar los márgenes de los elementos que queremos que sean mayor que cero.

Hágalo usted mismo: para crear la hoja de estilo para el documento del Listado 4-20, tiene que incluir todas las reglas presentadas desde el Listado 4-21, una sobre otra en el mismo archivo (misestilos.css).

Lo básico: lo que hemos hecho con la regla del Listado 4-21 es resetear los estilos de nuestro documento. Esta es una práctica común y generalmente requiere algo más que modificar los márgenes y rellenos, como hemos hecho en este ejemplo. Debido a que los estilos que debemos modificar son en la mayoría de los casos los mismos para cada proyecto, los desarrolladores han creado hojas de estilo que ya incluyen estas reglas y que podemos implementar en nuestros documentos junto con nuestros propios estilos. Estas hojas de estilo se denominan *hojas de estilo de reseteo (Reset*

Style Sheets) y hay varias disponibles. Para ver un ejemplo, visite meyerweb.com/eric/tools/css/reset/.

Con esta simple regla logramos que todos los elementos queden alineados a la izquierda de la ventana del navegador, sin márgenes alrededor y separados por la misma distancia entre ellos.

Este es el título
Principal
Fotos
Videos
Contacto
Título Primer Artículo
23
Viernes
Este es el texto de mi primer artículo

Figura 4-19: *Documento con estilos universales*

El siguiente paso es diseñar la cabecera. En este caso queremos que el elemento **<header>** se extienda hasta los límites de la ventana del navegador y su contenido esté centrado y se ubique dentro de un área no superior a 960 píxeles (este es un tamaño estándar para las pantallas anchas que tienen los ordenadores de escritorio). Las siguientes son las reglas que se requieren para este propósito.

```
#cabeceralogo {
  width: 96%;
  height: 150px;
  padding: 0% 2%;
  background-color: #0F76A0;
}
#cabeceralogo > div {
  width: 960px;
  margin: 0px auto;
  padding-top: 45px;
}
#cabeceralogo h1 {
  font: bold 54px Arial, sans-serif;
  color: #FFFFFF;
}
```

Listado 4-22: *Asignando estilos a la cabecera*

Como queremos que la cabecera tenga la anchura de la ventana, tenemos que declarar su tamaño en porcentaje. Cuando el tamaño de un elemento se declara en porcentaje, el navegador se encarga de calcular el tamaño real en píxeles a partir del tamaño actual de su contenedor (en este caso, la ventana del navegador). Para nuestro documento, queremos que la cabecera tenga el mismo ancho que la ventana, pero que incluya un relleno a los lados de modo que su contenido esté separado del borde. Con este propósito, asignamos un valor de 96 % a la propiedad **width** y declaramos un relleno de 2 % a los lados. Si la ventana tiene un ancho de 1000 píxeles, por ejemplo, el elemento **<header>** tendrá un ancho de 960 píxeles y un relleno de 20 píxeles a los lados.

Lo básico: si quiere asignar un margen o un relleno fijo a un elemento pero al mismo tiempo ajustar su ancho para que abarque todo el espacio disponible, puede asignar el valor `auto` a la propiedad `width`. Este valor le pide al navegador que calcule el ancho del elemento a partir del ancho de su contenedor, pero considerando los valores de los márgenes, los rellenos, y los bordes del elemento. Por ejemplo, si la ventana tiene un ancho de 1000 píxeles y asignamos a la cabecera un relleno de 50 píxeles a cada lado y el valor `auto` para su ancho, el navegador le otorgará un ancho de 900 píxeles.

La segunda regla en este ejemplo afecta a los elementos `<div>` que son descendientes directos del elemento `<header>`. Como solo tenemos un único elemento `<div>` que usamos para envolver el contenido de la cabecera, este es el elemento que modificará la regla. Las propiedades asignan un ancho de 960 píxeles y un margen con un valor de 0 píxeles para la parte superior e inferior, y el valor `auto` para el lado izquierdo y derecho. El valor `auto` le pide al navegador que calcule el margen de acuerdo con el tamaño del elemento y el espacio disponible en su contenedor. Esto hace que el navegador centre el elemento `<div>` y, por lo tanto, su contenido, cuando el ancho del contenedor es superior a 960 píxeles.

Finalmente, declaramos la regla `#cabeceralogo h1` para cambiar el tipo de letra y el color del elemento `<h1>` dentro de la cabecera. El resultado se muestra en la Figura 4-20.

Figura 4-20: *Cabecera*

Lo básico: el límite de 960 píxeles es un valor estándar que se usa para declarar el tamaño de páginas web para las pantallas anchas de los ordenadores personales y portátiles. El valor fue establecido considerando la capacidad de las personas para leer textos extensos. Para que un texto sea legible, se recomienda que tenga un máximo de 50-75 caracteres por línea. Si extendemos todo el contenido hasta los lados de la ventana del navegador, nuestro sitio web no se podría leer en pantallas anchas. Con la introducción de los dispositivos móviles, las limitaciones y requerimientos han cambiado. Los sitios web ya no se desarrollan con diseños fijos. Estudiaremos cómo crear diseños flexibles en la próxima sección de este capítulo y cómo adaptar nuestros sitios web a los dispositivos móviles usando diseño web adaptable en el Capítulo 5.

Al igual que la cabecera, el menú de nuestro diseño se extiende hasta los límites de la ventana, pero las opciones tienen que centrarse en un espacio no superior a 960 píxeles.

```
#menuprincipal {
  width: 96%;
```

```
  height: 50px;
  padding: 0% 2%;
  background-color: #9FC8D9;
  border-top: 1px solid #094660;
  border-bottom: 1px solid #094660;
}
#menuprincipal > div {
  width: 960px;
  margin: 0px auto;
}
```

Listado 4-23: *Asignando estilos al área de navegación*

Estos estilos posicionan el elemento `<nav>` y su contenido, pero los elementos `<ul>` y `<li>` que conforman el menú todavía tienen asignados los estilos por defecto que crean una lista vertical de ítems, y lo que necesitamos es colocar las opciones una al lado de la otra. Existen varias formas de organizar el elemento horizontalmente, pero la mejor alternativa, en este caso, es declarar los elementos `<li>` como elementos **`inline-block`**. De este modo, podemos posicionarlos en la misma línea y asignarles un tamaño personalizado.

```
#menuprincipal li {
  display: inline-block;
  height: 35px;
  padding: 15px 10px 0px 10px;
  margin-right: 5px;
}
#menuprincipal li:hover {
  background-color: #6FACC6;
}
#menuprincipal a {
  font: bold 18px Arial, sans-serif;
  color: #333333;
  text-decoration: none;
}
```

Listado 4-24: *Asignando estilos a las opciones del menú*

La primera regla del Listado 4-24 declara los elementos `<li>` dentro del elemento `<nav>` como elementos **`inline-block`** y les da una altura de 35 píxeles, con un relleno superior de 15 píxeles y 10 píxeles a los lados. Este ejemplo también incluye una regla con la seudoclase **`:hover`** para cambiar el color de fondo del elemento `<li>` cada vez que el ratón se encuentra sobre una opción. En la última regla, los elementos `<a>` también se modifican con un color y tipo de letra diferente. El resultado se muestra en la Figura 4-21.

Figura 4-21: *Menú*

Lo básico: una lista de ítems creada con los elementos **<ul>** y **<li>** tiene asignado por defecto el valor **list-item** para la propiedad **display**. Este modo crea una lista vertical de ítems con gráficos del lado izquierdo que los identifica. Cuando declara un valor diferente para la propiedad **display**, estos gráficos se eliminan. Para modificar o eliminar los indicadores cuando el modo es **list-item**, CSS ofrece la propiedad **list-style** (esta es una propiedad general que define los valores de las propiedades **list-style-image**, **list-style-position**, y **list-style-type**). CSS incluye múltiples valores que podemos asignar a esta propiedad para determinar el tipo de gráfico a mostrar. Los más usados son **none**, **square**, **circle**, y **decimal**. La propiedad también nos permite declarar la posición del gráfico (**inside** o **outside**) e incluso una imagen personalizada (por ejemplo, **list-style: url("migrafico.jpg");**).

A continuación, tenemos que diseñar la sección principal de nuestra página web. Esta sección se ha identificado con el elemento **<main>** y contiene los elementos **<section>** y **<aside>** que necesitamos convertir en columnas. Para seguir el mismo patrón de diseño usado para la cabecera y el menú, tenemos que extender el elemento **<main>** hasta los lados de la ventana y centrar su contenido.

```
main {
  width: 96%;
  padding: 2%;
  background-image: url("fondo.png");
}
main > div {
  width: 960px;
  margin: 0px auto;
}
```

Listado 4-25: *Asignando estilos al contenido principal*

Las reglas del Listado 4-25 asignan una imagen de fondo al elemento **<main>** para diferenciar el área principal del resto de la página. También agregamos un relleno del 2 % en la parte superior e inferior, de modo que el contenido del área principal se separa del menú y el pie de página.

Lo básico: los fondos cubren el área ocupada por el elemento y su relleno. Por defecto, se considera que los márgenes se encuentran fuera del elemento y, por lo tanto, no se ven afectados por la propiedad **background**. Si quiere que el fondo se muestre en toda el área ocupada por el elemento, tiene que evitar usar márgenes y en su lugar asignar rellenos, como hemos hecho en el Listado 4-25.

Con el área principal ya definida, es hora de crear las dos columnas que presentan el contenido principal de nuestra página web.

```
#articulosprincipales {
  float: left;
  width: 620px;
```

```
  padding-top: 30px;
  background-color: #FFFFFF;
  border-radius: 10px;
}
#infoadicional {
  float: right;
  width: 280px;
  padding: 20px;
  background-color: #E7F1F5;
  border-radius: 10px;
}
#infoadicional h1 {
  font: bold 18px Arial, sans-serif;
  color: #333333;
  margin-bottom: 15px;
}
.recuperar {
  clear: both;
}
```

Listado 4-26: *Creando las columnas para el contenido principal*

Es este caso, usamos la propiedad **float** para mover hacia los lados los elementos que representan cada columna. El elemento **<section>** se ha movido hacia la izquierda y el elemento **<aside>** hacia la derecha, dejando un espacio entre medio de 20 píxeles.

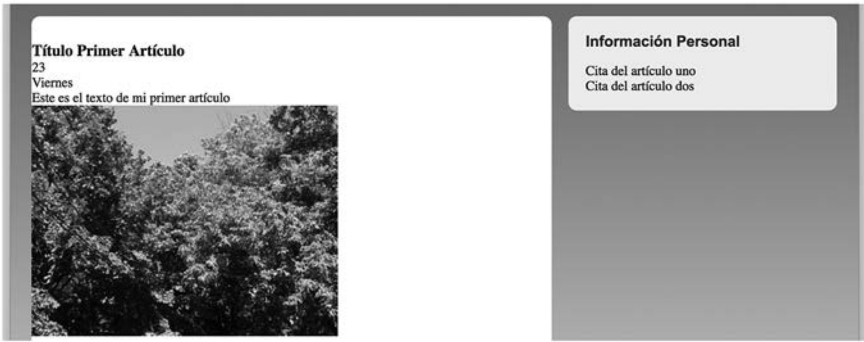

Figura 4-22: *Contenido principal*

Lo básico: cada vez que los elementos se posicionan con la propiedad **float** debemos acordarnos de recuperar el flujo normal del documento con el elemento siguiente. Con este propósito, en nuestro ejemplo agregamos un elemento **<div>** sin contenido debajo del elemento **<aside>**.

Ahora que las columnas están listas, tenemos que diseñar sus contenidos. El código del Listado 4-26 ya incluye una regla que configura el contenido del elemento **<aside>**, pero aún tenemos que configurar los elementos **<article>** en la primera columna.

Cada elemento **<article>** incluye un elemento **<time>** que representa la fecha en la que el artículo se ha publicado. Para nuestro diseño, decidimos mostrar esta fecha en una caja del lado izquierdo del artículo, por lo que este elemento debe tener una posición absoluta.

```
article {
  position: relative;
  padding: 0px 40px 20px 40px;
}
article time {
  display: block;
  position: absolute;
  top: -5px;
  left: -70px;
  width: 80px;
  padding: 15px 5px;

  background-color: #094660;
  box-shadow: 3px 3px 5px rgba(100, 100, 100, 0.7);
  border-radius: 5px;
}
.numerodia {
  font: bold 36px Verdana, sans-serif;
  color: #FFFFFF;
  text-align: center;
}
.nombredia {
  font: 12px Verdana, sans-serif;
  color: #FFFFFF;
  text-align: center;
}
```

Listado 4-27: *Configurando la posición y los estilos del elemento* `<time>`

Para asignar una posición absoluta al elemento **`<time>`**, tenemos que declarar la posición de su elemento padre como relativa. Esto lo logramos asignando el valor **`relative`** a la propiedad **`position`** en la primera regla del Listado 4-27. La segunda regla define la posición y el tamaño del elemento **`<time>`**, el cual se ubicará a 5 píxeles de la parte superior del elemento **`<article>`** y a 110 píxeles de su lado izquierdo.

Figura 4-23: *Posición absoluta para el elemento* `<time>`

El resto de las reglas tiene que asignar los estilos requeridos por los elementos restantes dentro de cada elemento **`<article>`**.

```
article h1 {
  margin-bottom: 5px;
  font: bold 30px Georgia, sans-serif;
}
article p {
  font: 18px Georgia, sans-serif;
}
figure {
  margin: 10px 0px;
}
figure img {
  padding: 5px;
  border: 1px solid;
}
```

Listado 4-28: *Asignando estilos a los artículos*

Figura 4-24: *Artículos*

Finalmente, tenemos que agregar unas pocas reglas a nuestra hoja de estilo para configurar el pie de página. El elemento **<footer>**, así como hemos hecho con el resto de los elementos estructurales, se debe extender hasta los lados de la ventana, pero en este caso el contenido se divide con tres elementos **<section>** para presentar la información en columnas. Aunque podríamos crear estas columnas con la propiedad **columns**, debido a que los elementos representan secciones de nuestro documento, la propiedad **float** es más apropiada.

```
#pielogo {
  width: 96%;
  padding: 2%;
  background-color: #0F76A0;
}
#pielogo > div {
  width: 960px;
  margin: 0px auto;
  background-color: #9FC8D9;
  border-radius: 10px;
}
.seccionpie {
  float: left;
  width: 270px;
  padding: 25px;
}
```

```
.seccionpie h1 {
  font: bold 20px Arial, sans-serif;
}
.seccionpie p {
  margin-top: 5px;
}
.seccionpie a {
  font: bold 16px Arial, sans-serif;
  color: #666666;
  text-decoration: none;
}
```

Listado 4-29: *Asignando estilos al pie de página*

Las reglas del Listado 4-29 definen tres secciones de 276 píxeles cada una, todas flotando hacia la izquierda para formar las tres columnas necesarias para nuestro diseño.

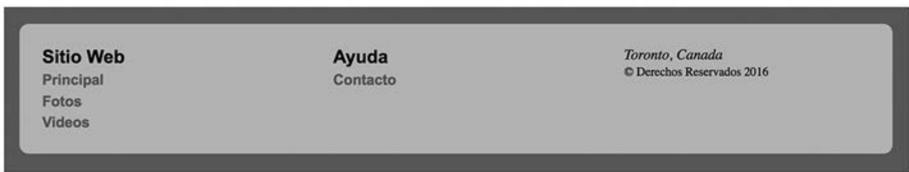

Figura 4-25: *Pie de página*

En estos casos, donde todas las secciones son iguales, en lugar de declarar los tamaños en píxeles es mejor hacerlo en porcentajes. El valor en porcentaje indica cuánto espacio va a ocupar el elemento dentro del contenedor. Por ejemplo, podemos declarar el ancho de las secciones en el pie de página con un valor de 27.33 % cada una y completar el 100 % con el relleno.

```
.seccionpie {
  float: left;
  width: 27.33%;
  padding: 3%;
}
```

Listado 4-30: *Calculando el tamaño de las secciones con valores en porcentajes*

Cuando los valores se declaran en porcentajes, el navegador se encarga de calcular cuántos píxeles tienen que asignarse a cada elemento de acuerdo con el espacio disponible en el contenedor. Si aplicamos la regla del Listado 4-30 a nuestro documento, el pie de página se mostrará parecido al de la Figura 4-25, pero los tamaños de las secciones los calculará el navegador antes de presentar la página (960 x 27.33 / 100 = 262).

IMPORTANTE: los valores en porcentaje también pueden ser utilizados para crear diseños flexibles, como veremos en el Capítulo 5, pero existe un modelo mejor para este fin llamado modelo de caja flexible. Estudiaremos el modelo de caja flexible en la siguiente sección de este capítulo.

Con estas reglas hemos finalizado el diseño del documento, pero esta es solo una de las páginas de nuestro sitio web. Este documento representa la página inicial, generalmente

almacenada en el archivo por defecto, como index.html, pero todavía tenemos que crear el resto de los documentos para representar cada página disponible. Afortunadamente, esta no es una tarea complicada. El mismo diseño que hemos desarrollado para la página inicial generalmente se comparte en todo el sitio web, solo tenemos que cambiar el área del contenido principal, pero el resto, como la cabecera, el pie de página y la estructura del documento en general son iguales, incluida la hoja de estilo, que normalmente comparten la mayoría de los documentos.

Hágalo usted mismo: si aún no lo ha hecho, asigne el nombre index.html al documento del Listado 4-20. Cree los archivos fotos.html, videos.html y contacto.html con una copia de este documento. Reemplace el contenido de los elementos **`<section>`** y **`<aside>`** en el área principal con el contenido correspondiente a cada página. Si es necesario, agregue nuevas reglas al archivo misestilos.css para ajustar el diseño de cada página. Aunque se recomienda concentrar todos los estilos en un solo archivo, también puede crear diferentes hojas de estilo por cada documento si lo considera adecuado. Abra el archivo index.html en su navegador y haga clic en los enlaces para navegar a través de las páginas.

Si necesitamos definir otras áreas en una página o dividir el área principal en secciones más pequeñas, podemos organizar las cajas con la propiedad **`float`**, como lo hemos hecho con las secciones dentro del pie de página. Aunque también podemos diseñar algunas áreas con posicionamiento absoluto o relativo, estos modos se reservan para posicionar contenido no relevante, como hemos hecho con las fechas de los artículos en nuestro ejemplo o para mostrar contenido oculto después de recibir una solicitud por parte del usuario. Por ejemplo, podemos crear un menú desplegable que muestra un submenú para cada opción. El siguiente ejemplo ilustra cómo agregar un submenú a la opción **Fotos** de nuestro documento.

```
<ul id="listamenu">
  <li><a href="index.html">Principal</a></li>
  <li><a href="fotos.html">Fotos</a>
    <ul>
      <li><a href="familia.html">Familia</a></li>
      <li><a href="vacaciones.html">Vacaciones</a></li>
    </ul>
  </li>
  <li><a href="videos.html">Videos</a></li>
  <li><a href="contacto.html">Contacto</a></li>
</ul>
```

Listado 4-31: *Agregando submenús*

Los estilos para un menú que incluye submenús difieren un poco del ejemplo anterior. Tenemos que listar las opciones del submenú de forma vertical en lugar de horizontal, posicionar la lista debajo de la barra del menú con valores absolutos, y solo mostrarla cuando el usuario mueve el ratón sobre la opción principal. Para este propósito, en el código del Listado 4-31, hemos agregado el identificador **`listamenu`** al elemento **`<ul>`** que representa el menú principal. Ahora, podemos diferenciar este elemento de los elementos **`<ul>`** encargados de representar los submenús (solo uno en nuestro ejemplo). Las siguientes reglas usan este identificador para asignar estilos a todos los menús y sus opciones.

```
#listamenu > li {
  position: relative;
  display: inline-block;
  height: 35px;
  padding: 15px 10px 0px 10px;
  margin-right: 5px;
}
#listamenu li > ul {
  display: none;
  position: absolute;
  top: 50px;
  left: 0px;

  background-color: #9FC8D9;
  box-shadow: 3px 3px 5px rgba(100, 100, 100, 0.7);
  border-radius: 0px 0px 5px 5px;

  list-style: none;
  z-index: 1000;
}
#listamenu li > ul > li {
  width: 120px;
  height: 35px;
  padding-top: 15px;
  padding-left: 10px;
}
#listamenu li:hover ul {
  display: block;
}
#listamenu a {
  font: bold 18px Arial, sans-serif;
  color: #333333;
  text-decoration: none;
}
#menuprincipal li:hover {
  background-color: #6FACC6;
}
```

Listado 4-32: *Asignando estilos a los submenús*

Las primeras dos reglas configuran las opciones principales y sus submenús. En estas reglas, asignamos una posición relativa a los elementos **<li>** que representan las opciones principales y luego especificamos una posición absoluta para los submenús de 50 píxeles debajo de la parte superior del menú. Esto posiciona los submenús debajo de la barra del menú. Para volver los submenús invisibles, declaramos la propiedad **display** con el valor **none**. También hemos tenido que incluir otras propiedades en esta regla como **list-style**, para eliminar los gráficos mostrados por defecto al lado izquierdo de las opciones, y la propiedad **z-index** para asegurarnos de que el submenú siempre se muestra sobre el resto de los elementos del área.

Como hemos hecho anteriormente, para responder al ratón tenemos que implementar la seudoclase **:hover**, pero este caso es algo diferente. Tenemos que mostrar el elemento **<ul>** que representa el submenú cada vez que el usuario mueve el ratón sobre cualquiera de las opciones principales, pero las opciones principales se representan con elementos **<li>**. La solución es

declarar la seudoclase **`:hover`** para los elementos **`<li>`**, pero agregar el nombre **`ul`** al final del selector para asignar los estilos a los elementos **`<ul>`** que se encuentran dentro del elemento **`<li>`** afectado por la seudoclase (**`#listamenu li:hover ul`**). Cuando el usuario mueve el ratón sobre el elemento **`<li>`** que representa una opción del menú principal, el valor **`block`** se asigna a la propiedad **`display`** del elemento **`<ul>`** y el submenú se muestra en la pantalla.

Figura 4-26: *Submenús*

Hágalo usted mismo: actualice su documento con el código del Listado 4-31 y las reglas de su hoja de estilo con el código del Listado 4-32. Abra el archivo index.html en su navegador y mueva el ratón sobre la opción **Fotos**. Debería ver algo similar a la Figura 4-26.

4.3 Modelo de caja flexible

El objetivo principal de un modelo de caja es el de ofrecer un mecanismo con el que dividir el espacio disponible en la ventana, y crear las filas y columnas que son parte del diseño de una página web. Sin embargo, las herramientas que ofrece el modelo de caja tradicional no cumplen este objetivo. Definir cómo distribuir las cajas y especificar sus tamaños, por ejemplo, no se puede lograr de forma eficiente con este modelo.

La dificultad en la implementación de patrones de diseño comunes, como expandir columnas para ocupar el espacio disponible, centrar el contenido de una caja en el eje vertical o extender una columna desde la parte superior a la parte inferior de una página, independiente del tamaño de su contenido, han forzado a la industria a buscar otras alternativas. Se han desarrollado varios modelos de caja, pero ninguno ha recibido más atención que el modelo de caja flexible.

IMPORTANTE: aunque el modelo de caja flexible tiene sus ventajas sobre el modelo de caja tradicional, se encuentra aún en estado experimental y algunos navegadores no pueden procesar sus propiedades. Esta es la razón por la que en este capítulo también introducimos el modelo de caja tradicional. Antes de elegir el modelo a aplicar en sus sitios web debe considerar estas limitaciones. Para determinar si puede usar cualquiera de las herramientas que introduce HTML5, visite www.caniuse.com.

Contenedor flexible

El modelo de caja flexible resuelve los problemas del modelo de caja tradicional de una manera elegante. Este modelo aprovecha las herramientas que usa el modelo de caja tradicional, como el posicionamiento absoluto y las columnas, pero en lugar de hacer flotar los

elementos organiza las cajas usando contenedores flexibles. Un contenedor flexible es un elemento que convierte su contenido en cajas flexibles. En este nuevo modelo, cada grupo de cajas debe estar incluido dentro de otra caja que es la encargada de configurar sus características, tal como muestra el siguiente ejemplo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <section id="cajapadre">
    <div id="caja-1">Caja 1</div>
    <div id="caja-2">Caja 2</div>
    <div id="caja-3">Caja 3</div>
    <div id="caja-4">Caja 4</div>
  </section>
</body>
</html>
```

Listado 4-33: *Organizando cajas con un contenedor flexible*

El documento del Listado 4-33, al igual que los ejemplos anteriores, incluye un elemento **`<section>`** que actúa como contenedor de otros elementos. La diferencia se presenta en cómo se configuran los elementos desde CSS. Para volver flexibles las cajas dentro del elemento **`<section>`** (sus tamaños cambian de acuerdo al espacio disponible), tenemos que convertir a este elemento en un contendor flexible. Para este propósito, CSS ofrece los valores **`flex`** e **`inline-flex`** para la propiedad **`display`**. El valor **`flex`** define un elemento Block flexible y el valor **`inline-flex`** define un elemento Inline flexible.

```
#cajapadre {
  display: flex;
}
```

Listado 4-34: *Convirtiendo el elemento* `cajapadre` *en un contenedor flexible*

Hágalo usted mismo: cree un nuevo archivo HTML con el código del Listado 4-33 y un archivo CSS llamado misestilos.css con la regla del Listado 4-34. Abra el documento en su navegador. Debería ver las cajas dentro del elemento **`<section>`** una al lado de la otra en la misma línea.

Elementos flexibles

Para que un elemento dentro de un contenedor flexible se vuelva flexible, tenemos que declararlo como tal. Las siguientes son las propiedades disponibles para configurar elementos flexibles.

flex-grow—Esta propiedad declara la proporción en la cual el elemento se va a expandir o encoger. La proporción se determina considerando los valores asignados al resto de los elementos de la caja (los elementos hermanos).

flex-shrink—Esta propiedad declara la proporción en la que el elemento se va a reducir. La proporción se determina a partir de los valores asignados al resto de los elementos de la caja (los elementos hermanos).

flex-basis—Esta propiedad declara un tamaño inicial para el elemento.

flex—Esta propiedad nos permite configurar los valores de las propiedades **`flex-grow`**, **`flex-shrink`**, y **`flex-basis`** al mismo tiempo.

Las cajas flexibles se expanden o encogen para ocupar el espacio libre dentro de la caja padre. La distribución del espacio depende de las propiedades del resto de las cajas. Si todas las cajas se configuran como flexibles, el tamaño de cada uno de ellas dependerá del tamaño de la caja padre y del valor de la propiedad **`flex`**.

```
#cajapadre {
  display: flex;
  width: 600px;
}
#cajapadre > div {
  height: 145px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  flex: 1;
}
#caja-2 {
  flex: 1;
}
#caja-3 {
  flex: 1;
}
#caja-4 {
  flex: 1;
}
```

Listado 4-35: *Convirtiendo las cajas en cajas flexibles con la propiedad* `flex`

En el ejemplo del Listado 4-35, solo declaramos el valor **`flex-grow`** para la propiedad **`flex`** con el fin de determinar cómo se expandirán las cajas. El tamaño de cada caja se calcula multiplicando el valor del tamaño de la caja padre por el valor de su propiedad **`flex`** dividido por la suma de los valores **`flex-grow`** de todas las cajas. Por ejemplo, la fórmula para el elemento **`caja-1`** es **600 × 1 / 4 = 150**. El valor **600** es el tamaño de la caja padre, **1** es el valor de la propiedad **`flex`** asignado al elemento **`caja-1`**, y **4** es la suma de los valores de la propiedad **`flex`** asignados a cada una de las cajas. Debido a que todas las cajas de nuestro ejemplo tienen el mismo valor **1** en su propiedad **`flex`**, el tamaño de cada caja será de 150 píxeles menos los márgenes (hemos asignado un margen de 5 píxeles al elemento **`<div>`**).

Figura 4-27: *Los mismos valores para la propiedad* `flex` *distribuyen el espacio equitativamente*

Hágalo usted mismo: reemplace las reglas en su archivo CSS por el código del Listado 4-35 y abra el documento del Listado 4-33 en su navegador. Debería ver algo parecido a la Figura 4-27.

El potencial de esta propiedad es evidente cuando asignamos diferentes valores a cada elemento.

```
#cajapadre {
  display: flex;
  width: 600px;
}
#cajapadre > div {
  height: 145px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  flex: 2;
}
#caja-2 {
  flex: 1;
}
#caja-3 {
  flex: 1;
}
#caja-4 {
  flex: 1;
}
```

Listado 4-36: *Creando una distribución desigual*

En el Listado 4-36, asignamos el valor **2** a la propiedad **`flex`** del elemento **`caja-1`**. Ahora, la fórmula para calcular el tamaño de esta caja es **600 × 2 / 5 = 240**. Debido a que no cambiamos el tamaño de la caja padre, el primer valor de la fórmula es el mismo, pero el segundo valor es **2** (el nuevo valor de la propiedad **`flex`** de **`caja-1`**), y la suma de los valores de todas las propiedades **`flex`** es **5** (**2** para **`caja-1`** y **1** para cada una de las otras tres cajas). Aplicando la misma fórmula para el resto de las cajas, podemos obtener sus tamaños: **600 × 1 / 5 = 120**.

Figura 4-28: *Distribución desigual con* `flex`

Comparando los resultados, podemos ver cómo se distribuye el espacio. El espacio disponible se divide en porciones de acuerdo a la suma de los valores de la propiedad **`flex`** de cada caja (un total de **5** en nuestro ejemplo). Luego, las porciones se distribuyen entre las cajas. El elemento **`caja-1`** recibe dos porciones porque el valor de su propiedad **`flex`** es **`2`** y el resto de elementos recibe solo una porción porque el valor de sus propiedades **`flex`** es **`1`**. La ventaja no es solo que los elementos se vuelven flexibles, sino también que cada vez que agregamos un nuevo elemento, no tenemos que calcular su tamaño; los tamaños de todas las cajas se recalculan automáticamente.

Estas características son interesantes, pero existen otros escenarios posibles. Por ejemplo, cuando una de las cajas es inflexible y tiene un tamaño explícito, las otras cajas se flexionan para compartir el resto del espacio disponible.

```
#cajapadre {
  display: flex;
  width: 600px;
}
#cajapadre > div {
  height: 145px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  width: 300px;
}
#caja-2 {
  flex: 1;
}

#caja-3 {
  flex: 1;
}
#caja-4 {
  flex: 1;
}
```

Listado 4-37: *Combinando cajas flexibles con cajas inflexibles*

La primera caja del ejemplo del Listado 4-37 tiene un tamaño de 300 píxeles, por lo que el espacio a distribuir entre el resto de las cajas es de 300 píxeles (**600 – 300 = 300**). El navegador calcula el tamaño de cada caja flexible con la misma fórmula que hemos usado anteriormente: **300 × 1 / 3 = 100**.

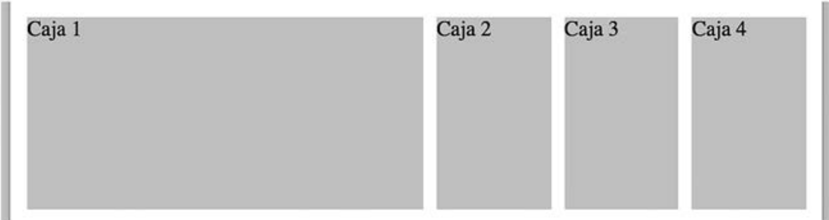

Figura 4-29: *Solo se distribuye el espacio libre*

Del mismo modo que podemos tener una caja con un tamaño explícito, podemos tener dos o más. El principio es el mismo, solo que el espacio remanente se distribuye entre las cajas flexibles.

Existe un posible escenario en el cual podríamos tener que declarar el tamaño de un elemento pero mantenerlo flexible. Para lograr esta configuración, debemos utilizar el resto de los parámetros disponibles para la propiedad **`flex`** (**`flex-shrink`** y **`flex-basis`**).

```
#cajapadre {
  display: flex;
}
#cajapadre > div {
  height: 145px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  flex: 1 1 200px;
}
#caja-2 {
  flex: 1 5 100px;
}
#caja-3 {
  flex: 1 5 100px;
}
#caja-4 {
  flex: 1 5 100px;
}
```

Listado 4-38: *Controlando cómo se encoge el elemento*

Esta vez, se han declarado tres parámetros para la propiedad **`flex`** de cada caja. El primer parámetro de todas las cajas (**`flex-grow`**) se ha definido con el valor `1`, declarando la misma proporción de expansión. La diferencia se determina por el agregado de los valores para los parámetros **`flex-shrink`** y **`flex-basis`**. El parámetro **`flex-shrink`** trabaja como **`flex-grow`**, pero determina la proporción en la que las cajas se reducirán para caber en el espacio disponible. En nuestro ejemplo, el valor de este parámetro es `1` para el elemento **`caja-1`** y `5` para el resto de las cajas, lo cual asignará más espacio a **`caja-1`**. El parámetro **`flex-basis`**, por otro lado, establece un valor inicial para el elemento. Por lo tanto, el valor del parámetro **`flex-basis`** se considera para calcular cuánto se expandirá o reducirá un elemento flexible. Cuando este valor es 0 o no se declara, el valor considerado es el tamaño del contenido del elemento.

Hágalo usted mismo: reemplace las reglas de su archivo CSS por el código del Listado 4-38. En este ejemplo, no declaramos el tamaño del elemento **`cajapadre`,** de modo que cuando la ventana del navegador se expande, el elemento padre también se expande y todas las cajas de su interior aumentan en la misma proporción. Por el contrario, cuando el tamaño de la ventana se reduce, el elemento **`caja-1`** se reduce en una proporción diferente debido al valor especificado para el parámetro **`flex-shrink`** (1 en lugar de 5).

IMPORTANTE: el valor 0 asignado a los parámetros **`flex-grow`** o **`flex-shrink`** no permite que el elemento se expanda o reduzca, respectivamente. Para declarar flexibilidad, los valores de estos parámetros deben ser iguales o mayores que 1.

También podemos asignar el valor **`auto`** al parámetro **`flex-basis`** para pedirle al navegador que use el valor de la propiedad **`width`** como referencia.

```
#cajapadre {
  display: flex;
  width: 600px;
}
#cajapadre > div {
  height: 145px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  width: 200px;
  flex: 1 1 auto;
}
#caja-2 {
  width: 100px;
  flex: 1 1 auto;
}
#caja-3 {
  width: 100px;
  flex: 1 1 auto;
}
#caja-4 {
  width: 100px;
  flex: 1 1 auto;
}
```

Listado 4-39: *Definiendo cajas flexibles con un tamaño preferido*

En el Listado 4-39, cada caja tiene un ancho preferido (**`width`**), pero después de que todas las cajas quedan posicionadas hay un espacio libre de 100 píxeles. Este espacio extra se dividirá entre las cajas flexibles. Para calcular la porción de espacio asignado a cada caja, usamos la misma fórmula que antes: **100 × 1 / 4 = 25**. Esto significa que se agregan 25 píxeles adicionales al tamaño preferido de cada caja (menos los márgenes).

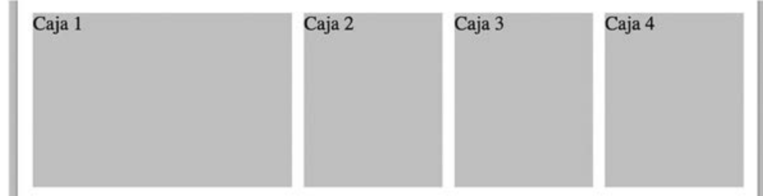

Figura 4-30: *Espacio libre distribuido entre las cajas*

La propiedad **`flex-basis`** nos permite definir un tamaño inicial para influenciar al navegador a la hora de distribuir el espacio disponible entre las cajas, pero las cajas aún se expanden o reducen más allá de ese valor. CSS ofrece las siguientes propiedades para poner limitaciones al tamaño de una caja.

max-width—Esta propiedad especifica el ancho máximo permitido para el elemento. Acepta valores en cualquiera de las unidades disponibles en CSS, como píxeles o porcentajes.

min-width—Esta propiedad especifica el ancho mínimo permitido para el elemento. Acepta valores en cualquiera de las unidades disponibles en CSS, como píxeles o porcentajes.

max-height—Esta propiedad especifica la altura máxima permitida para el elemento. Acepta valores en cualquiera de las unidades disponibles en CSS, como píxeles o porcentajes.

min-height—Esta propiedad especifica la altura mínima permitida para el elemento. Acepta valores en cualquiera de las unidades disponibles en CSS, como píxeles o porcentajes.

El siguiente ejemplo declara todas las cajas como flexibles, pero asigna un ancho máximo de 50 píxeles al elemento **`caja-1`**. Independientemente del espacio disponible, **`caja-1`** nunca supera los 50 píxeles.

```
#cajapadre {
  display: flex;
}
#cajapadre > div {
  height: 145px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  max-width: 50px;
  flex: 1;
}
#caja-2 {
  flex: 1;
}
#caja-3 {
  flex: 1;
}
#caja-4 {
  flex: 1;
}
```

Listado 4-40: *Declarando un tamaño máximo*

Figura 4-31: *Cajas con un tamaño máximo*

Organizando elementos flexibles

Por defecto, los elementos dentro de un contenedor flexible se muestran horizontalmente en la misma línea, pero no se organizan con una orientación estándar. Un contenedor flexible usa ejes para describir la orientación de su contenido. La especificación declara dos ejes que son independientes de la orientación: el eje principal y el eje transversal. El eje principal es aquel en el que se presenta el contenido (normalmente es equivalente a la orientación horizontal), y el eje transversal es el perpendicular al eje principal (normalmente equivalente a la orientación vertical). Si la orientación cambia, los ejes se desplazan junto con el contenido.

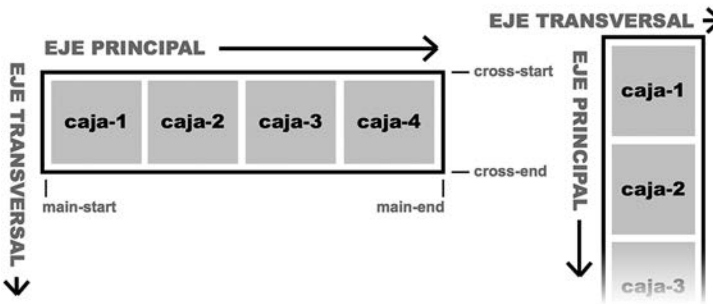

Figura 4-32: *Ejes de contenedores flexibles*

Las propiedades definidas para este modelo trabajan con estos ejes y organizan los elementos desde sus extremos: main-start, main-end, cross-start y cross-end. La relación entre estos extremos es similar a la relación entre los extremos izquierdo y derecho, o superior e inferior usados para describir la dirección horizontal y vertical en modelos convencionales, pero en este modelo esa relación se invierte cuando la orientación cambia. Cuando uno de estos extremos, como main-start, se menciona en la descripción de una propiedad, debemos recordar que puede referirse al extremo izquierdo o superior, dependiendo de la orientación actual del contenedor (en el diagrama izquierdo de la Figura 4-32, por ejemplo, el extremo main-start está referenciando el lado izquierdo del contenedor, mientras que en el diagrama de la derecha referencia el extremo superior).

Una vez que entendamos cómo trabaja con este modelo, podemos cambiar la organización de las cajas. CSS ofrece las siguientes propiedades con este propósito.

flex-direction—Esta propiedad define el orden y la orientación de las cajas en un contenedor flexible. Los valores disponibles son `row`, `row-reverse`, `column` y `column-reverse`, con el valor `row` configurado por defecto.

order—Esta propiedad especifica el orden de las cajas. Acepta números enteros que determinan la ubicación de cada caja.

justify-content—Esta propiedad determina cómo se va a distribuir el espacio libre. Los valores disponibles son `flex-start`, `flex-end`, `center`, `space-between`, y `space-around`.

align-items—Esta propiedad alinea las cajas en el eje transversal. Los valores disponibles son `flex-start`, `flex-end`, `center`, `baseline`, y `stretch`.

align-self—Esta propiedad alinea una caja en el eje transversal. Trabaja como `align-items` pero afecta cajas de forma individual. Los valores disponibles son `auto`, `flex-start`, `flex-end`, `center`, `baseline`, y `stretch`.

flex-wrap—Esta propiedad determina si se permiten crear múltiples líneas de cajas. Los valores disponibles son `nowrap`, `wrap`, y `wrap-reverse`.

align-content—Esta propiedad alinea las líneas de cajas en el eje vertical. Los valores disponibles son `flex-start`, `flex-end`, `center`, `space-between`, `space-around`, y `stretch`.

Si lo que necesitamos es configurar la dirección de las cajas, podemos usar la propiedad `flex-direction`. Esta propiedad se asigna al contenedor con un valor que corresponde al orden que queremos otorgar al contenido. El valor `row` declara la orientación de las cajas de acuerdo a la orientación del texto (normalmente horizontal) y ordena las cajas desde main-start a main-end (normalmente de izquierda a derecha). El valor `row-reverse` declara la misma orientación que `row`, pero invierte el orden de los elementos desde main-end a main-start (normalmente de derecha a izquierda). El valor `column` declara la orientación de acuerdo a la orientación en la cual se presentan los bloques de texto (normalmente vertical) y ordena las cajas desde main-start a main-end (normalmente de extremo superior a inferior). Y finalmente, el valor `column-reverse` declara la misma orientación que `column`, pero invierte el orden de los elementos desde main-end a main-start (normalmente de extremo inferior a superior). El siguiente ejemplo revierte el orden natural de una línea de cajas.

```
#cajapadre {
  display: flex;
  flex-direction: row-reverse;
}
#cajapadre > div {
  height: 145px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  flex: 1;
}
#caja-2 {
  flex: 1;
}
#caja-3 {
  flex: 1;
}
#caja-4 {
  flex: 1;
}
```

Listado 4-41: *Invirtiendo la orientación de las cajas*

Figura 4-33: *Cajas en orden invertido*

Lo básico: CSS ofrece una propiedad llamada **`writing-mode`** que determina la orientación de las líneas de texto (horizontal o vertical), y esta es la razón por la cual el resultado de las propiedades del modelo de caja flexible siempre depende de la orientación establecida previamente para el texto. Para obtener más información acerca de esta propiedad, visite nuestro sitio web y siga los enlaces de este capítulo.

El orden de las cajas también se puede personalizar. La propiedad **`order`** nos permite declarar la ubicación de cada caja.

```
#cajapadre {
  display: flex;
}
#cajapadre > div {
  height: 145px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  flex: 1;
  order: 2;
}
#caja-2 {
  flex: 1;
  order: 4;
}
#caja-3 {
  flex: 1;
  order: 3;
}
#caja-4 {
  flex: 1;
  order: 1;
}
```

Listado 4-42: *Definiendo la posición de cada caja*

Figura 4-34: *Nuevas posiciones para cada caja definidas por la propiedad* `order`

Una característica importante del modelo de caja flexible es la capacidad de distribuir el espacio libre. Cuando las cajas no ocupan todo el espacio en el contenedor, dejan espacio libre que debe ubicarse en alguna parte del diseño. Por ejemplo, las siguientes reglas declaran un tamaño de 600 píxeles para el contenedor flexible y un ancho de 100 píxeles para cada caja, dejando un espacio libre de 200 píxeles (menos los márgenes).

```
#cajapadre {
  display: flex;
  width: 600px;
  border: 1px solid;
}
#cajapadre > div {
  height: 145px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  width: 100px;
}
#caja-2 {
  width: 100px;
}
#caja-3 {
  width: 100px;
}
#caja-4 {
  width: 100px;
}
```

Listado 4-43: *Distribuyendo el espacio libre en un contenedor flexible*

El ejemplo del Listado 4-43 agrega un borde al contenedor para poder identificar el espacio extra. Por defecto, las cajas se ordenan desde main-start a main-end (normalmente de izquierda a derecha), dejando un espacio libre al final.

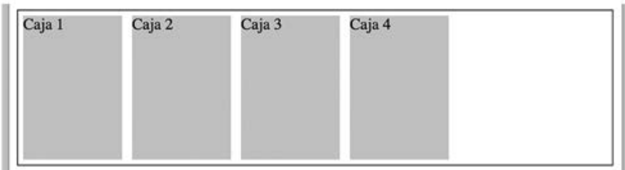

Figura 4-35: *Cajas y espacio libre dentro de un contendor flexible*

Este comportamiento se puede modificar con la propiedad **`justify-content`**. El valor por defecto asignado a esta propiedad es **`flex-start`**, que ordena las cajas según se muestra en la Figura 4-35, aunque podemos asignar un valor diferente para personalizar la forma en la que se distribuyen las cajas y el espacio libre. Por ejemplo, el valor **`flex-end`** desplaza el espacio al comienzo del contenedor y las cajas hacia el final.

```
#cajapadre {
  display: flex;
```

```
  width: 600px;
  border: 1px solid;
  justify-content: flex-end;
}
#cajapadre > div {
  height: 145px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  width: 100px;
}
#caja-2 {
  width: 100px;
}
#caja-3 {
  width: 100px;
}
#caja-4 {
  width: 100px;
}
```

Listado 4-44: *Distribuyendo el espacio vacío con la propiedad* `justify-content`

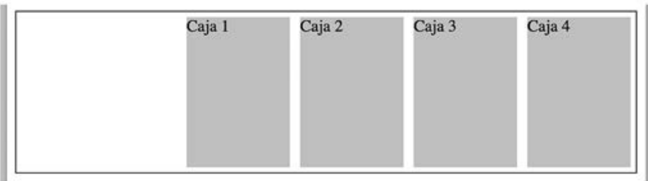

Figura 4-36: *Espacio vacío distribuido con* `justify-content: flex-end`

Las siguientes figuras muestran el efecto que produce el resto de los valores disponibles para esta propiedad.

Figura 4-37: *Espacio vacío distribuido con* `justify-content: center`

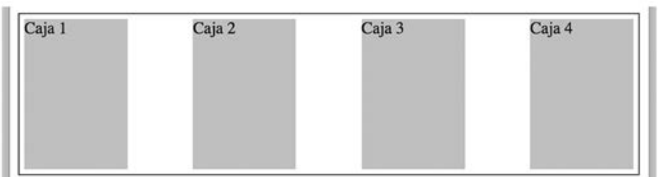

Figura 4-38: *Espacio vacío distribuido con* `justify-content: space-between`

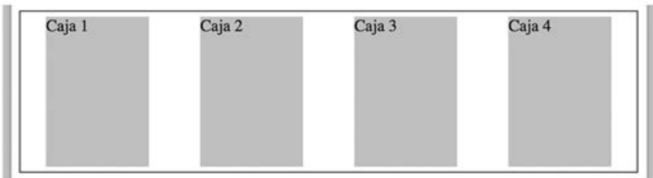

Figura 4-39: *Espacio vacío distribuido con* `justify-content: space-around`

Otra propiedad que puede ayudarnos a distribuir espacio es **`align-items`**. Esta propiedad trabaja como **`justify-content`** pero alinea las cajas en el eje transversal. Esta característica logra que la propiedad sea apropiada para realizar una alineación vertical.

```
#cajapadre {
  display: flex;
  width: 600px;
  height: 200px;
  border: 1px solid;
  align-items: center;
}
#cajapadre > div {
  height: 145px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  flex: 1;
}
#caja-2 {
  flex: 1;
}
#caja-3 {
  flex: 1;
}
#caja-4 {
  flex: 1;
}
```

Listado 4-45: *Distribuyendo el espacio vertical*

En el Listado 4-45 hemos definido la altura del contenedor, dejando un espacio libre de 55 píxeles. Debido a que asignamos el valor **`center`** a la propiedad **`align-items`**, este espacio se distribuye hacia el extremo superior e inferior, tal como muestra la Figura 4-40.

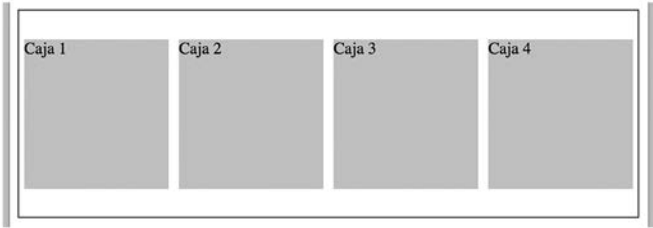

Figura 4-40: *Alineación vertical con* `align-items: center`

Los valores disponibles para la propiedad **`align-items`** son **`flex-start`**, **`flex-end`**, **`center`**, **`baseline`** y **`stretch`**. El último valor estira las cajas desde el extremo superior al inferior para adaptarlas al espacio disponible. Esta característica es tan importante que el valor **`stretch`** se declara por defecto para todos los contenedores flexibles. El efecto que logra el valor **`stretch`** es que, cada vez que no se declara la altura de las cajas, estas adoptan automáticamente el tamaño de sus elementos padre.

```
#cajapadre {
  display: flex;
  width: 600px;
  height: 200px;
  border: 1px solid;
  align-items: stretch;
}
#cajapadre > div {
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  flex: 1;
}
#caja-2 {
  flex: 1;
}
#caja-3 {
  flex: 1;
}
#caja-4 {
  flex: 1;
}
```

Listado 4-46: *Estirando las cajas para ocupar el espacio vertical disponible*

Figura 4-41: *Estirando las cajas con* `align-items: stretch`

Esta característica es extremadamente útil cuando nuestro diseño presenta columnas con diferente contenido. Si dejamos que el contenido determine la altura, una columna será más corta que la otra. Asignando el valor **`stretch`** a la propiedad **`align-items`**, las columnas más cortas se estiran para coincidir con las más largas.

Esta propiedad también ofrece el valor **`flex-start`** para alinear las cajas al comienzo de la línea, el cual queda determinado por la orientación del contenedor (normalmente el extremo izquierdo o superior).

```
#cajapadre {
  display: flex;
  width: 600px;
  height: 200px;
  border: 1px solid;
  align-items: flex-start;
}
#cajapadre > div {
  height: 145px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  flex: 1;
}
#caja-2 {
  flex: 1;
}
#caja-3 {
  flex: 1;
}
#caja-4 {
  flex: 1;
}
```

Listado 4-47: *Alineando las cajas hacia el extremo superior*

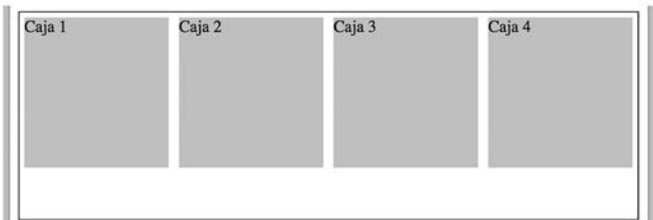

Figura 4-42: *Cajas alineadas con* `align-items: flex-start`

El valor **`flex-end`** alinea las cajas hacia el final del contenedor (normalmente el extremo derecho o inferior).

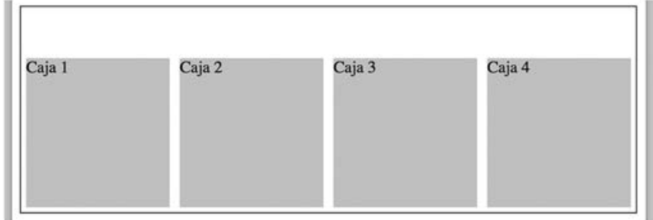

Figura 4-43: *Cajas alineadas con* `align-items: flex-end`

Finalmente, el valor **baseline** alinea las cajas por la línea base de la primera línea de contenido. El siguiente ejemplo asigna un tipo de letra diferente al contenido del elemento **caja-2** para mostrar el efecto producido por este valor.

```
#cajapadre {
  display: flex;
  width: 600px;
  height: 200px;
  border: 1px solid;
  align-items: baseline;
}
#cajapadre > div {
  height: 145px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  flex: 1;
}
#caja-2 {
  flex: 1;
  font-size: 36px;
}
#caja-3 {
  flex: 1;
}
#caja-4 {
  flex: 1;
}
```

Listado 4-48: *Alineando las cajas por la línea base*

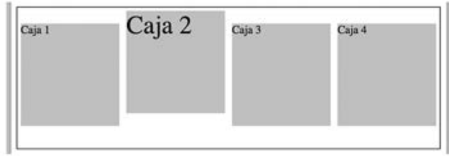

Figura 4-44: *Cajas alineadas con* `align-items: baseline`

A veces puede resultar útil alinear las cajas de forma independiente, sin importar la alineación establecida por el contenedor. Esto se puede lograr asignando la propiedad **align-self** a la caja que queremos modificar.

```
#cajapadre {
  display: flex;
  width: 600px;
  height: 200px;
  border: 1px solid;
  align-items: flex-end;
}
#cajapadre > div {
  height: 145px;
```

```
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  flex: 1;
}
#caja-2 {
  flex: 1;
  align-self: center;
}
#caja-3 {
  flex: 1;
}
#caja-4 {
  flex: 1;
}
```

Listado 4-49: *Cambiando la alineación del elemento* `caja-2`

Las reglas del Listado 4-49 alinean los elementos hacia el extremo inferior del contenedor, excepto el elemento **`caja-2`**, que se alinea hacia el centro por la propiedad **`align-self`**.

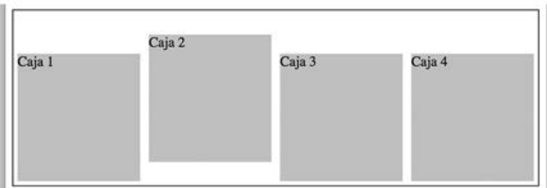

Figura 4-45: *Caja alineada con* `align-self`

Un contenedor flexible puede organizar las cajas en una o varias líneas. La propiedad **`flex-wrap`** declara esta condición usando tres valores: **`nowrap`**, **`wrap`** y **`wrap-reverse`**. El valor **`nowrap`** define el contenedor flexible como un contenedor de una sola línea (las líneas no se agrupan), el valor **`wrap`** define el contenedor como un contenedor de múltiples líneas y las ordena desde el extremo cross-start a cross-end, mientras que el valor **`wrap-reverse`** genera múltiples líneas en orden invertido.

```
#cajapadre {
  display: flex;
  width: 600px;
  border: 1px solid;
  justify-content: center;
  flex-wrap: wrap;
}
#cajapadre > div {
  height: 40px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  width: 100px;
}
```

```
#caja-2 {
  width: 100px;
}
#caja-3 {
  width: 100px;
}
#caja-4 {
  flex: 1 1 400px;
}
```

Listado 4-50: *Creando dos líneas de cajas con la propiedad* `flex-wrap`

En el Listado 4-50, las primeras tres cajas tienen un tamaño de 100 píxeles, suficiente como para ubicarlas en una sola línea dentro de un contenedor de 600 píxeles de ancho, pero la última caja se declara como flexible con un tamaño inicial de 400 píxeles (**`flex-basis`**) y, por lo tanto, no hay espacio suficiente en el contendor para ubicar todas las cajas en una sola línea. El navegador tiene dos opciones: puede reducir el tamaño de la caja flexible para ubicarla en el espacio disponible o generar una nueva línea. Debido a que la propiedad **`flex-wrap`** se ha declarado con el valor **`wrap`**, se crea una nueva línea, como en la Figura 4-46.

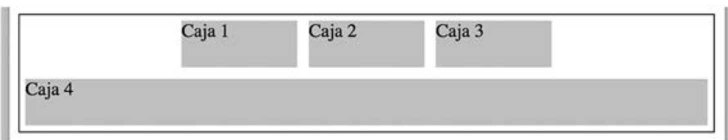

Figura 4-46: *Múltiples líneas en un contenedor flexible*

El elemento **`caja-4`** se ha declarado como flexible por la propiedad **`flex`**, por lo que no solo se ubica en una nueva línea, sino que también se expande hasta ocupar todo el espacio disponible (el valor de 400 píxeles declarado por el parámetro **`flex-basis`** es solo el ancho sugerido, no una declaración de tamaño). Aprovechando el espacio libre que deja la última caja, las tres primeras cajas quedan alineadas con la propiedad **`justify-content`**.

El orden de las líneas se puede invertir con el valor **`wrap-reverse`**, tal como se ilustra a continuación.

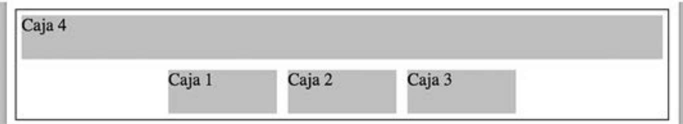

Figura 4-47: *Nuevo ordenamiento de líneas usando* `flex-wrap: wrap-reverse`

Cuando tenemos contenedores con múltiples líneas, es posible que necesitemos alinearlas. CSS ofrece la propiedad **`align-content`** para alinear líneas de cajas en un contenedor flexible.

```
#cajapadre {
  display: flex;
  width: 600px;
  height: 200px;
  border: 1px solid;
  flex-wrap: wrap;
```

```
  align-content: flex-start;
}
#cajapadre > div {
  height: 50px;
  margin: 5px;
  background-color: #CCCCCC;
}
#caja-1 {
  flex: 1 1 100px;
}
#caja-2 {
  flex: 1 1 100px;
}
#caja-3 {
  flex: 1 1 100px;
}
#caja-4 {
  flex: 1 1 400px;
}
```

Listado 4-51: *Alineando múltiples líneas con la propiedad* `align-content`

La propiedad **`align-content`** puede tener seis valores: **`flex-start`**, **`flex-end`**, **`center`**, **`space-between`**, **`space-around`** y **`stretch`**. El valor **`stretch`** se declara por defecto y lo que hace es expandir las líneas para llenar el espacio disponible a menos que se haya declarado un tamaño fijo para los elementos.

En el ejemplo del Listado 4-51, todas las cajas se declaran como flexibles con un ancho y una altura inicial de 50 píxeles, y el elemento **`cajapadre`** se define como un contenedor de múltiples líneas con la propiedad **`flex-wrap`**. Esto crea un contenedor flexible con dos líneas, similar al del ejemplo anterior, pero con espacio vertical suficiente para experimentar.

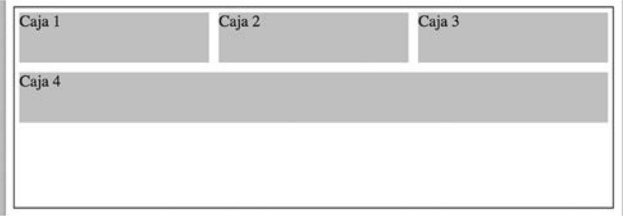

Figura 4-48: *Líneas alineadas con* `align-content: flex-start`

Las siguientes figuras muestran el efecto producido por el resto de los valores disponibles para la propiedad **`align-content`**.

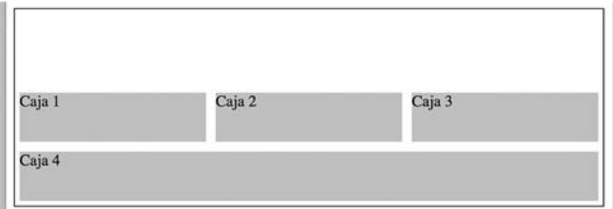

Figura 4-49: *Líneas alineadas con* `align-content: flex-end`

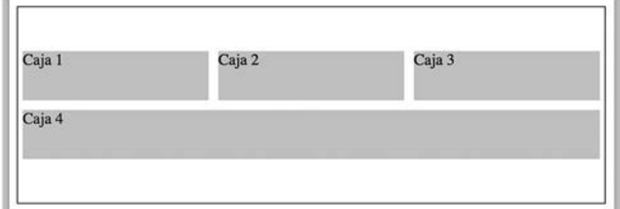

Figura 4-50: *Líneas alineadas con* `align-content: center`

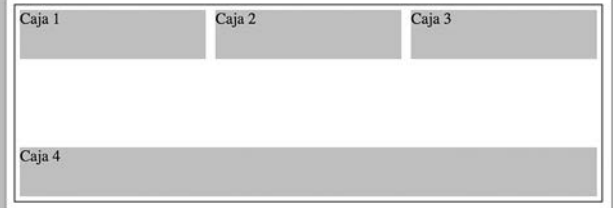

Figura 4-51: *Líneas alineadas con* `align-content: space-between`

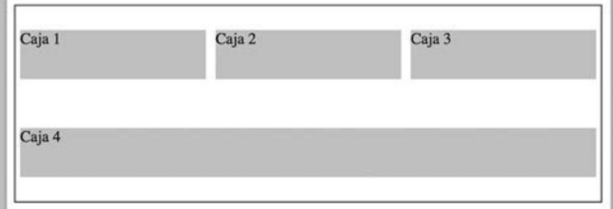

Figura 4-52: *Líneas alineadas con* `align-content: space-around`

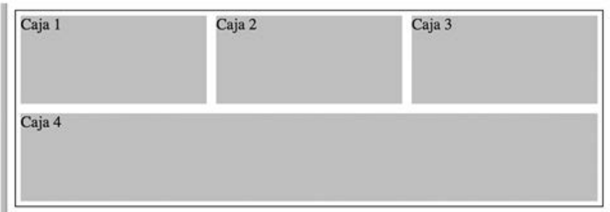

Figura 4-53: *Líneas alineadas con* `align-content: stretch`

Aplicación de la vida real

No existe mucha diferencia entre un documento diseñado con el modelo de caja tradicional y el que tenemos que usar para implementar el modelo de caja flexible. Por ejemplo, para diseñar una página web con el modelo de caja flexible a partir del documento del Listado 4-20, solo tenemos que eliminar los elementos **`<div>`** que hemos para recuperar el flujo normal del documento. El resto del documento permanece igual.

```
<!DOCTYPE html>
<html lang="es">
<head>
```

```
    <title>Este texto es el título del documento</title>
    <meta charset="utf-8">
    <meta name="description" content="Este es un documento HTML5">
    <meta name="keywords" content="HTML, CSS, JavaScript">
    <link rel="stylesheet" href="misestilos.css">
  </head>
  <body>
    <header id="cabeceralogo">
      <div>
        <h1>Este es el título</h1>
      </div>
    </header>
    <nav id="menuprincipal">
      <div>
        <ul>
          <li><a href="index.html">Principal</a></li>
          <li><a href="fotos.html">Fotos</a></li>
          <li><a href="videos.html">Videos</a></li>
          <li><a href="contacto.html">Contacto</a></li>
        </ul>
      </div>
    </nav>
    <main>
      <div>
        <section id="articulosprincipales">
          <article>
            <h1>Título Primer Artículo</h1>
            <time datetime="2016-12-23" pubdate>
              <div class="numerodia">23</div>
              <div class="nombredia">Viernes</div>
            </time>
            <p>Este es el texto de mi primer artículo</p>
            <figure>
              <img src="miimagen.jpg">
            </figure>
          </article>
          <article>
            <h1>Título Segundo Artículo</h1>
            <time datetime="2016-12-7" pubdate>
              <div class="numerodia">7</div>
              <div class="nombredia">Miércoles</div>
            </time>
            <p>Este es el texto de mi segundo artículo</p>
            <figure>
              <img src="miimagen.jpg">
            </figure>
          </article>
        </section>
        <aside id="infoadicional">
          <h1>Información Personal</h1>
          <p>Cita del artículo uno</p>
          <p>Cita del artículo dos</p>
        </aside>
      </div>
    </main>
    <footer id="pielogo">
```

```
    <div>
      <section class="seccionpie">
        <h1>Sitio Web</h1>
        <p><a href="index.html">Principal</a></p>
        <p><a href="fotos.html">Fotos</a></p>
        <p><a href="videos.html">Videos</a></p>
      </section>

      <section class="seccionpie">
        <h1>Ayuda</h1>
        <p><a href="contacto.html">Contacto</a></p>
      </section>
      <section class="seccionpie">
        <address>Toronto, Canada</address>
        <small>&copy; Derechos Reservados 2016</small>
      </section>
    </div>
  </footer>
</body>
</html>
```

Listado 4-52: *Definiendo un documento para aplicar el modelo de caja flexible*

Como la organización de los elementos estructurales es la misma, las reglas CSS que tenemos que aplicar al documento también son similares. Solo tenemos que transformar los elementos estructurales en contenedores flexibles y volver flexible su contenido cuando el diseño lo demanda. Por ejemplo, las siguientes reglas asignan el modo **flex** para la propiedad **display** del elemento **<header>** y declaran al elemento **<div>** dentro de este elemento como flexible, para que la cabecera y su contenido se expandan hasta ocupar el espacio disponible en la ventana.

```
* {
  margin: 0px;
  padding: 0px;
}
#cabeceralogo {
  display: flex;
  justify-content: center;
  width: 96%;
  height: 150px;
  padding: 0% 2%;
  background-color: #0F76A0;
}
#cabeceralogo > div {
  flex: 1;
  max-width: 960px;
  padding-top: 45px;
}
#cabeceralogo h1 {
  font: bold 54px Arial, sans-serif;
  color: #FFFFFF;
}
```

Listado 4-53: *Convirtiendo la cabecera en un contenedor flexible*

Como queremos que el contenido de la cabecera quede centrado en la pantalla, tenemos que asignar el valor **center** a la propiedad **justify-content**. El elemento **<div>** se ha declarado flexible, lo que significa que se expandirá o reducirá de acuerdo con el espacio disponible, pero, como hemos explicado antes, cuando la página se presenta en dispositivos con pantalla ancha, tenemos que asegurarnos de que no sea demasiado ancho y al usuario le resulte incómodo leer su contenido. Esto se resuelve con la propiedad **max-width**. Gracias a esta propiedad, el elemento **<div>** no se expandirá más de 960 píxeles.

El mismo procedimiento se debe aplicar a nuestro menú, pero el resto de los elementos dentro del elemento **<nav>** usan las mismas propiedades y valores implementados anteriormente.

```
#menuprincipal {
  display: flex;
  justify-content: center;
  width: 96%;
  height: 50px;
  padding: 0% 2%;
  background-color: #9FC8D9;
  border-top: 1px solid #094660;
  border-bottom: 1px solid #094660;
}
#menuprincipal > div {
  flex: 1;
  max-width: 960px;
}
#menuprincipal li {
  display: inline-block;
  height: 35px;
  padding: 15px 10px 0px 10px;
  margin-right: 5px;
}
#menuprincipal li:hover {
  background-color: #6FACC6;
}
#menuprincipal a {
  font: bold 18px Arial, sans-serif;
  color: #333333;
  text-decoration: none;
}
```

Listado 4-54: *Convirtiendo el menú en un contenedor flexible*

Las reglas para el contenido principal también son las mismas, pero esta vez tenemos dos contenedores flexibles, uno representado por el elemento **<main>** responsable de centrar el contenido en la página, y otro representado por el elemento **<div>** encargado de configurar las dos columnas creadas por los elementos **<section>** y **<aside>**. Por esta razón, la regla que afecta al elemento **<div>** requiere ambas propiedades: **display** para declarar el elemento como un contenedor flexible y **flex** para declarar el contenedor mismo como un elemento flexible.

```
main {
  display: flex;
  justify-content: center;
```

```
  width: 96%;
  padding: 2%;
  background-image: url("fondo.png");
}
main > div {
  display: flex;
  flex: 1;
  max-width: 960px;
}
#articulosprincipales {
  flex: 1;
  margin-right: 20px;
  padding-top: 30px;
  background-color: #FFFFFF;
  border-radius: 10px;
}
#infoadicional {
  width: 280px;
  padding: 20px;
  background-color: #E7F1F5;
  border-radius: 10px;
}
#infoadicional h1 {
  font: bold 18px Arial, sans-serif;
  color: #333333;
  margin-bottom: 15px;
}
```

Listado 4-55: *Convirtiendo las columnas en contenedores flexibles*

El elemento **<aside>** se ha declarado con un ancho fijo, lo que significa que solo la columna de la izquierda, representada por el elemento **<section>**, se va a expandir o reducir para ocupar el resto del espacio disponible.

Las propiedades para el contenido del elemento **<section>** (la columna izquierda) son exactamente las mismas que las que hemos definido para el modelo de caja tradicional.

```
article {
  position: relative;
  padding: 0px 40px 20px 40px;
}
article time {
  display: block;
  position: absolute;
  top: -5px;
  left: -70px;
  width: 80px;
  padding: 15px 5px;
  background-color: #094660;
  box-shadow: 3px 3px 5px rgba(100, 100, 100, 0.7);
  border-radius: 5px;
}
.numerodia {
  font: bold 36px Verdana, sans-serif;
  color: #FFFFFF;
```

```
  text-align: center;
}
.nombredia {
  font: 12px Verdana, sans-serif;
  color: #FFFFFF;
  text-align: center;
}
article h1 {
  margin-bottom: 5px;
  font: bold 30px Georgia, sans-serif;
}
article p {
  font: 18px Georgia, sans-serif;
}
figure {
  margin: 10px 0px;
}
figure img {
  max-width: 98%;
  padding: 1%;
  border: 1px solid;
}
```

Listado 4-56: *Configurando el contenido*

El pie de página presenta un desafío similar al contenido principal. Tenemos que convertir al elemento **<footer>** en una caja flexible y declarar como flexibles las tres columnas creadas en su interior para presentar la información.

```
#pielogo {
  display: flex;
  justify-content: center;
  width: 96%;
  padding: 2%;
  background-color: #0F76A0;
}
#pielogo > div {
  display: flex;
  flex: 1;
  max-width: 960px;
  background-color: #9FC8D9;
  border-radius: 10px;
}
.seccionpie {
  flex: 1;
  padding: 25px;
}
.seccionpie h1 {
  font: bold 20px Arial, sans-serif;
}
.seccionpie p {
  margin-top: 5px;
}
```

```
.seccionpie a {
  font: bold 16px Arial, sans-serif;
  color: #666666;
  text-decoration: none;
}
```

Listado 4-57: *Convirtiendo al pie de página en un contenedor flexible*

Hágalo usted mismo: cree un nuevo archivo HTML con el código del Listado 4-52 y un archivo CSS llamado misestilos.css con las reglas introducidas desde el Listado 4-53. Recuerde incluir la imagen miimagen.jpg en el mismo directorio. Abra el documento en su navegador. Debería ver la misma página creada anteriormente con el modelo de caja tradicional, con la excepción de que esta vez las secciones de la página son flexibles (sus tamaños cambian a medida que el tamaño de la ventana se incrementa o reduce) y la columna generada por el elemento `<aside>` se extiende hasta el extremo inferior del área principal.

Capítulo 5
Diseño web adaptable

5.1 Web móvil

La introducción en el mercado del iPhone en el año 2007 cambió el mundo para siempre. Hasta ese momento, solo disponíamos de móviles con pantallas pequeñas y sin la capacidad suficiente de descargar y mostrar sitios web. Si queríamos navegar en la Web, teníamos que hacerlo en un ordenador personal. Esto presentaba una situación fácil de controlar para los desarrolladores. Los usuarios contaban con un único tipo de pantalla y esta tenía el tamaño suficiente para incluir todo lo que necesitaban. Las resoluciones más populares del momento incluían al menos 1024 píxeles horizontales, lo cual permitía a los desarrolladores diseñar sus sitios web con un tamaño estándar (como el que presentamos en el capítulo anterior). Pero cuando el iPhone apareció en el mercado, los sitios web con un diseño fijo como estos no se podían leer en una pantalla tan pequeña. La primera solución fue desarrollar dos sitios web separados, uno para ordenadores y otro para iPhones, pero la introducción en el mercado de nuevos dispositivos, como el iPad en el año 2010, forzó nuevamente a los desarrolladores a buscar mejores alternativas. Dar flexibilidad a los elementos fue una de las soluciones implementadas, pero no resultó suficiente. Las páginas web con varias columnas no se mostraban bien en pantallas pequeñas. La solución final debía incluir elementos flexibles, además de la posibilidad de adaptar el diseño a cada dispositivo. Así es como nació lo que hoy conocemos como diseño web adaptable, o por su nombre en inglés *responsive web design*.

El diseño web adaptable es una técnica que combina diseños flexibles con una herramienta provista por CSS llamada *Media Queries* (consulta de medios), que nos permite detectar el tamaño de la pantalla y realizar los cambios necesarios para adaptar el diseño a cada situación.

Media Queries

Una Media Query es una regla reservada en CSS que se incorporó con el propósito de permitir a los desarrolladores detectar el medio en el que se muestra el documento. Por ejemplo, usando Media Queries podemos determinar si el documento se muestra en un monitor o se envía a una impresora, y asignar los estilos apropiados para cada caso. Para este propósito, las Media Queries ofrecen las siguientes palabras clave.

all—Las propiedades se aplican en todos los medios.

print—Las propiedades se aplican cuando la página web se envía a una impresora.

screen—Las propiedades se aplican cuando la página web se muestra en una pantalla color.

speech—Las propiedades se aplican cuando la página web se procesa por un sintetizador de voz.

Lo que hace que las Media Queries sean útiles para el diseño web es que también pueden detectar el tamaño del medio. Con las Media Queries podemos detectar el tamaño del área de visualización (la parte de la ventana del navegador donde se muestran nuestras páginas web) y definir diferentes reglas CSS para cada dispositivo. Existen varias palabras clave que podemos usar para detectar estas características. Las siguientes son las que más se usan para desarrollar un sitio web con diseño web adaptable.

width—Esta palabra clave determina el ancho en que se aplican las propiedades.

height—Esta palabra clave determina la altura a la que se aplican las propiedades.

min-width—Esta palabra clave determina el ancho mínimo desde el cual que se aplican las propiedades.

max-width—Esta palabra clave determina el ancho máximo hasta el cual que se aplican las propiedades.

aspect-ratio—Esta palabra clave determina la proporción en la cual que se aplican las propiedades.

orientation—Esta palabra clave determina la orientación en la cual que se aplican las propiedades. Los valores disponibles son **`portrait`** (vertical) y **`landscape`** (horizontal).

resolution—Esta palabra clave determina la densidad de píxeles en la cual que se aplican las propiedades. Acepta valores en puntos por pulgada (dpi), puntos por centímetro (dpcm) o por proporción en píxeles (dppx). Por ejemplo, para detectar una pantalla de tipo retina con una escala de 2, podemos usar el valor **`2dppx`**.

Usando estas palabras clave, podemos detectar ciertos aspectos del medio y del área de visualización en los que se va a mostrar la página y modificar las propiedades para ajustar el diseño a las condiciones actuales. Por ejemplo, si definimos la Media Query con la palabra clave **`width`** y el valor **`768px`**, las propiedades solo se aplicarán cuando la página se muestra en un iPad estándar en modo portrait (orientación vertical).

Para definir una Media Query, podemos declarar solo las palabras clave y los valores que necesitamos. Las palabras clave que describen una característica, como el ancho del área de visualización, tienen que declararse entre paréntesis. Si se incluye más de una palabra clave, podemos asociarlas con los operadores lógicos **`and`** (y) y **`or`** (o). Por ejemplo, la Media Query **`all and (max-width: 480px)`** asigna las propiedades al documento en todos los medios, pero solo cuando el área de visualización tiene un ancho de 480 píxeles o menos.

Existen dos maneras de declarar Media Queries: desde el documento usando el atributo **`media`** del elemento **`<link>`**, o desde la hoja de estilo con la regla **`@media`**. Cuando usamos el elemento **`<link>`**, podemos seleccionar el archivo CSS con la hoja de estilo que queremos cargar para una configuración específica. Por ejemplo, el siguiente documento carga dos archivos CSS, uno que contiene los estilos generales que aplicaremos en toda situación y medios, y otro con los estilos requeridos para presentar la página en un dispositivo de pantalla pequeña (480 píxeles de ancho o menos).

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="stylesheet" href="adaptabletodos.css">
  <link rel="stylesheet" media="(max-width: 480px)"
href="adaptablecelulares.css">
</head>
<body>
```

```
  <section>
    <div>
      <article>
        <h1>Título Primer Artículo</h1>
        <img src="miimagen.jpg">
      </article>
    </div>
  </section>
  <aside>
    <div>
      <h1>Información</h1>
      <p>Cita del artículo uno</p>
      <p>Cita del artículo dos</p>
    </div>
  </aside>
  <div class="recuperar"></div>
</body>
</html>
```

Listado 5-1: *Cargando estilos con Media Queries*

Cuando el navegador lee este documento, primero carga el archivo adaptabletodos.css y luego, si el ancho del área de visualización es de 480 píxeles o inferior, carga el archivo adaptablecelulares.css y aplica los estilos al documento.

CSS son las iniciales del nombre hojas de estilo en cascada (*Cascading Style Sheets*), lo cual significa que las propiedades se procesan en cascada y, por lo tanto, las nuevas propiedades reemplazan a las anteriores. Cuando tenemos que modificar un elemento para un área de visualización particular, solo necesitamos declarar las propiedades que queremos cambiar, pero los valores del resto de las propiedades definidas anteriormente permanecen iguales. Por ejemplo, podemos asignar un color de fondo al cuerpo del documento en el archivo adaptabletodos.css y luego cambiarlo en el archivo acaptablecelulares.css, pero este nuevo valor solo se aplicará cuando el documento se presenta en una pantalla pequeña. La siguiente es la regla que implementaremos en el archivo adaptabletodos.css.

```
body {
  margin: 0px;
  padding: 0px;
  background-color: #990000;
}
```

Listado 5-2: *Definiendo los estilos por defecto (adaptabletodos.css)*

Los estilos definidos en el archivo adaptablecelulares.css tienen que modificar solo los valores de las propiedades que queremos cambiar (en este caso la propiedad **`background-color`**), pero el resto de los valores debe permanecer igual a como se han declarado en el archivo adaptabletodos.css.

```
body {
  background-color: #3333FF;
}
```

Listado 5-3: *Definiendo los estilos para pantallas pequeñas (adaptablecelulares.css)*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 5-1. Cree un archivo CSS llamado adaptabletodos.css con la regla del Listado 5-2 y un archivo CSS llamado adaptablecelulares.css con la regla del Listado 5-3. Abra el documento en su navegador. Debería ver la página con un fondo rojo. Arrastre un lado de la ventana para reducir su tamaño. Cuando el ancho del área de visualización sea de 480 píxeles o inferior, debe ver el color del fondo cambiar a azul.

Con el elemento **<link>** y el atributo **media** podemos cargar diferentes hojas de estilo para cada situación, pero cuando solo se modifican unas pocas propiedades, podemos incluir todas en la misma hoja de estilo y definir las Media Queries con la regla **@media**, tal como ilustra el siguiente ejemplo.

```
body {
  margin: 0px;
  padding: 0px;
  background-color: #990000;
}
@media (max-width: 480px) {
  body {
    background-color: #3333FF;
  }
}
```

Listado 5-4: *Declarando las Media Queries con la regla* @media *(*adaptabletodos.css*)*

La Media Query se declara con la regla **@media** seguida de las palabras clave y valores que definen las características del medio y las propiedades entre llaves. Cuando el navegador lee esta hoja de estilo, asigna las propiedades de la regla **body** al elemento **<body>** y luego, si el ancho del área de visualización es de 480 píxeles o inferior, cambia el color de fondo del elemento **<body>** a **#3333FF** (azul).

Hágalo usted mismo: reemplace las reglas en su archivo adaptabletodos.css por el código del Listado 5-4 y elimine el segundo elemento **<link>** del documento del Listado 5-1 (en este ejemplo, solo necesita un archivo CSS para todos sus estilos). Abra el documento en su navegador y reduzca el tamaño de la ventana para ver cómo CSS aplica la regla definida por la Media Query cuando el ancho del área de visualización es de 480 píxeles o menos.

Puntos de interrupción

En el ejemplo anterior, declaramos una Media Query que detecta si el ancho del área de visualización es igual o menor a 480 píxeles y aplica las propiedades si se satisface la condición. Esta es una práctica común. Los dispositivos disponibles en el mercado estos días presentan una gran variedad de pantallas de diferentes tamaños. Si usamos la palabra clave **width** para detectar un dispositivo específico, algunos dispositivos que no conocemos o se acaban de introducir en el mercado no se detectarán y mostrarán nuestro sitio web de forma incorrecta. Por ejemplo, la Media Query **width: 768px** detecta solo los iPads de tamaño estándar en modo portrait (vertical) porque solo esos dispositivos en esa orientación específica presentan

ese tamaño. Un iPad en modo landscape (horizontal), o un iPad Pro, que presenta una resolución diferente, o cualquier otro dispositivo con una pantalla más pequeña o más grande de 768 píxeles, no se verá afectado por estas propiedades, incluso aunque el tamaño varíe solo uno o dos píxeles. Para evitar errores, no debemos seleccionar tamaños específicos. En su lugar, debemos establecer puntos máximos o mínimos en los cuales el diseño debe cambiar significativamente, y utilizar el modelo de caja para adaptar el tamaño de los elementos al espacio disponible cuando el ancho de la pantalla se encuentra entre estos puntos. Estos puntos máximos y mínimos se especifican con las palabras clave **max-width** y **min-width**, y se llaman *puntos de interrupción* (*breakpoints* en inglés).

Los puntos de interrupción son aquellos en los que el diseño de una página web requiere cambios significativos para poder adaptarse al tamaño de la pantalla. Estos puntos son los tamaños en los cuales la flexibilidad de los elementos no es suficiente para ajustar las cajas al espacio disponible y tenemos que mover los elementos de un lugar a otro, o incluso excluirlos del diseño para que nuestro sitio web siga siendo legible. No existen estándares establecidos que podamos seguir para determinar estos puntos; todo depende de nuestro diseño. Por ejemplo, podríamos decidir que cuando el ancho del área de visualización es igual o menor que 480 píxeles tenemos que usar solo una columna para presentar la información, pero esto solo es posible si el diseño original contiene más de una columna. Los valores más comunes son 320, 480, 768, y 1024.

Para establecer un punto de interrupción, tenemos que declarar una Media Query con las palabras clave **max-width** o **min-width**, de modo que las propiedades se aplicarán cuando el tamaño del área de visualización supere estos límites. La palabra clave que aplicamos depende de la dirección que queramos seguir. Si queremos diseñar primero para dispositivos con pantalla pequeña, debemos establecer tamaños mínimos con **min-width**, pero si queremos establecer un diseño genérico aplicable a las pantallas anchas de ordenadores personales y luego modificar las propiedades a medida que el tamaño se reduce, lo mejor es establecer máximos con **max-width**. Por ejemplo, la siguiente hoja de estilo asigna un color de fondo por defecto al cuerpo del documento, pero a medida que el tamaño de la pantalla se reduce, el color se modifica.

```
body {
  margin: 0px;
  padding: 0px;
  background-color: #990000;
}
@media (max-width: 1024px) {
  body {
    background-color: #3333FF;
  }
}
@media (max-width: 768px) {
  body {
    background-color: #FF33FF;
  }
}
@media (max-width: 480px) {
  body {
    background-color: #339933;
  }
}
@media (max-width: 320px) {
  body {
```

```
    background-color: #CCCCCC;
  }
}
```

Listado 5-5: *Incluyendo múltiples puntos de interrupción*

Cuando se asignan los estilos del Listado 5-5 al documento del Listado 5-1, el navegador presenta la página con un fondo rojo en un área de visualización superior a 1024 píxeles, pero cambia de color cada vez que el ancho se encuentra por debajo de los límites establecidos por las Media Queries.

Hágalo usted mismo: reemplace las reglas del archivo adaptabletodos.css con el código del Listado 5-5. Abra el documento en su navegador y modifique el tamaño de la ventana. Si el área de visualización es superior a 1024 píxeles, el cuerpo se muestra con un fondo rojo, pero si el tamaño se reduce a 1024 píxeles o menos, el color cambia a azul, a 768 píxeles o menos, cambia a rosado, a 480 píxeles o menos cambia a verde, y a 320 píxeles o menos cambia a gris.

Área de visualización

El área de visualización (*viewport* en inglés) es la parte de la ventana del navegador donde se muestran nuestras páginas web. Debido a la relación entre la ventana y el área de visualización, ambos deberían presentar el mismo tamaño en dispositivos móviles, pero esto no se cumple en todos los casos. Por defecto, algunos dispositivos asignan un ancho de 980 píxeles al área de visualización, sin importar su tamaño real o el tamaño real de la pantalla. Esto significa que las Media Queries de nuestras hojas de estilo verán un ancho de 980 píxeles cuando en realidad el tamaño del área de visualización es totalmente diferente. Para normalizar esta situación y forzar al navegador a definir el tamaño del área de visualización igual al tamaño real de la pantalla, tenemos que declarar el elemento **<meta>** en la cabecera de nuestros documentos con el nombre **viewport** y valores que determinan el ancho y la escala que queremos ver. Los dos valores requeridos son **width** e **initial-scale** para declarar el ancho del área de visualización y su escala. El siguiente ejemplo ilustra cómo debemos configurar el elemento **<meta>** para pedirle al navegador que defina el tamaño y la escala reales de la pantalla del dispositivo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="adaptabletodos.css">
</head>
<body>
  <section>
    <div>
      <article>
```

```
        <h1>Título Primer Artículo</h1>
        <img src="miimagen.jpg">
      </article>
    </div>
  </section>
  <aside>
    <div>
      <h1>Información</h1>
      <p>Cita del artículo uno</p>
      <p>Cita del artículo dos</p>
    </div>
  </aside>
  <div class="recuperar"></div>
</body>
</html>
```

Listado 5-6: *Configurando el área de visualización con el elemento* `<meta>`

Lo básico: el elemento **`<meta>`** que declara el área de visualización puede incluir otros valores para configurar atributos como las escalas mínimas y máximas (**`minimum-scale`** y **`maximum-scale`**), o si queremos permitir a los usuarios ampliar o reducir la página web (**`user-scalable`**). Para más información, visite nuestro sitio web y siga los enlaces de este capítulo.

Flexibilidad

En un diseño web adaptable tenemos que permitir que nuestras página sean flexibles, de modo que los elementos se adapten al espacio disponible cuando el ancho del área de visualización se encuentra entre los puntos de interrupción. Esto se puede lograr con el modelo de caja flexible: solo tenemos que crear contenedores flexibles y declarar sus tamaños como flexibles con la propiedad **`flex`** (ver Capítulo 4), pero este modelo de caja no lo reconocen algunos navegadores y, por lo tanto, la mayoría de los desarrolladores aún implementan el modelo de caja tradicional. Aunque el modelo de caja tradicional no fue diseñado para trabajar con elementos flexibles, podemos volver flexibles los elementos declarando sus tamaños en porcentaje. Cuando declaramos el tamaño de un elemento en porcentaje, el navegador calcula el tamaño real en píxeles a partir del tamaño de su contenedor. Por ejemplo, si asignamos un ancho de 80 % a un elemento que se encuentra dentro de otro elemento con un tamaño de 1000 píxeles, el ancho del elemento será de 800 píxeles (80 % de 1000). Debido a que el tamaño de los elementos cambia cada vez que lo hace el tamaño de sus contenedores, estos se vuelven flexibles.

El siguiente ejemplo crea dos columnas con los elementos **`<section>`** y **`<aside>`** del documento del Listado 5-6, y declara sus anchos en porcentaje para hacerlos flexibles.

```
* {
  margin: 0px;
  padding: 0px;
}
section {
  float: left;
  width: 70%;
  background-color: #999999;
}
```

```
aside {
  float: left;
  width: 30%;
  background-color: #CCCCCC;
}
.recuperar {
  clear: both;
}
```

Listado 5-7: *Creando elementos flexibles con valores en porcentaje*

Las reglas del Listado 5-7 declaran el tamaño de los elementos **<section>** y **<aside>** al 70 % y al 30 % del tamaño de su contenedor, respetivamente. Si el tamaño de la pantalla cambia, el navegador recalcula el tamaño de los elementos y, por lo tanto, estos elementos se expanden o reducen de acuerdo con el espacio disponible.

Figura 5-1: *Elementos flexibles con el modelo de caja tradicional*

Cada vez que declaramos el ancho del elemento en porcentajes, tenemos que asegurarnos de que la suma total de los valores no exceda el 100 %. De otro modo, los elementos no entrarán en el espacio disponible y se moverán a una nueva línea. En el ejemplo del Listado 5-7, esto fue fácil de lograr porque solo teníamos dos elementos sin márgenes, rellenos o bordes, pero tan pronto como agregamos un valor adicional, como el relleno, tenemos que recordar restar estos valores al ancho del elemento o el total excederá el 100 %. Por ejemplo, las siguientes reglas reducen el ancho de los elementos **<section>** y **<aside>** para poder agregar un relleno de 2 % a cada lado y un margen de 5 % entre medio.

```
* {
  margin: 0px;
  padding: 0px;
}
section {
  float: left;
  width: 61%;
  padding: 2%;
  margin-right: 5%;
  background-color: #999999;
}
aside {
  float: left;
  width: 26%;
```

```
  padding: 2%;
  background-color: #CCCCCC;
}
.recuperar {
  clear: both;
}
```

Listado 5-8: *Agregando márgenes y relleno a elementos flexibles*

Las reglas del Listado 5-8 asignan al elemento **`<section>`** un ancho de 61 %, un relleno de 2 % del lado superior, derecho, inferior e izquierdo, y un margen de 5 % del lado derecho, y al elemento **`<aside>`** un ancho de 26 % y un relleno de 2 % del lado superior, derecho, inferior e izquierdo. Debido a que la suma de estos valores no excede el 100 % (61 + 2 + 2 + 5 + 26 + 2 + 2 = 100), los elementos se colocan lado a lado en la misma línea.

Figura 5-2: *Elementos flexibles con relleno y márgenes*

Hágalo usted mismo: reemplace las reglas del archivo adaptabletodos.css por el código del Listado 5-8. Incluya la imagen miimagen.jpg en el directorio del documento. Abra el documento en su navegador. Debería ver algo similar a la Figura 5-2.

Box-sizing

Cada vez que se calcula el total del área ocupada por un elemento, el navegador obtiene el valor final con la fórmula tamaño + márgenes + relleno + bordes. Si declaramos la propiedad **`width`** con un valor de 100 píxeles, **`margin`** a 20 píxeles, **`padding`** a 10 píxeles y **`border`** a 1 píxel, el total del área horizontal ocupada por el elemento será **`100 + 40 + 20 + 2 = 162`** píxeles (los valores de las propiedades **`margin`**, **`padding`** y **`border`** se han duplicado en la fórmula porque asumimos que se han asignado los mismos valores a los lados izquierdo y derecho de la caja). Esto significa que cada vez que declaramos el tamaño de un elemento con la propiedad **`width,`** tenemos que recordar que el área que se necesita en la página para mostrar el elemento puede ser mayor. Esta situación es particularmente problemática cuando los tamaños se definen en porcentajes, o cuando intentamos combinar valores en porcentaje con otros tipos de unidades como píxeles. CSS incluye la siguiente propiedad para modificar este comportamiento.

box-sizing—Esta propiedad nos permite decidir cómo se calculará el tamaño de un elemento y forzar a los navegadores a incluir el relleno y el borde en el tamaño declarado por la propiedad **`width`**. Los valores disponibles son **`content-box`** y **`border-box`**.

Por defecto, el valor de la propiedad **`box-sizing`** se declara como **`content-box`**, lo cual significa que el navegador agregará los valores de las propiedades **`padding`** y **`border`** al tamaño especificado por la propiedad **`width`**. Si en cambio asignamos a esta propiedad el valor **`border-box`**, el navegador incluye el relleno y el borde como parte del ancho, y entonces disponemos de otras opciones como combinar anchos definidos en porcentaje con rellenos en píxeles, tal como se muestra en el siguiente ejemplo.

```
* {
  margin: 0px;
  padding: 0px;
}
section {
  float: left;
  width: 65%;
  padding: 20px;
  margin-right: 5%;
  background-color: #999999;
  box-sizing: border-box;
}
aside {
  float: left;
  width: 30%;
  padding: 20px;
  background-color: #CCCCCC;
  box-sizing: border-box;
}
.recuperar {
  clear: both;
}
```

Listado 5-9: *Incluyendo el relleno y el borde en el tamaño del elemento*

En el Listado 5-9 declaramos la propiedad **`box-sizing`** con el valor **`border-box`** para ambos elementos, **`<section>`** y **`<aside>`**. Debido a esto, ya no tenemos que considerar el relleno cuando calculamos el tamaño de los elementos (65 + 5 + 30 = 100).

Hágalo usted mismo: reemplace las reglas del archivo adaptabletodos.css con el código del Listado 5-9. Abra el documento en su navegador. Debería ver algo similar a la Figura 5-2, pero esta vez el relleno no cambia cuando se modifica el tamaño de la ventana porque su valor se ha definido en píxeles.

Fijo y flexible

En los anteriores ejemplos hemos visto cómo definir cajas flexibles con rellenos fijos, pero muchas veces nos encontraremos con la necesidad de tener que combinar columnas completas con valores flexibles y fijos. Nuevamente, esto es muy fácil de lograr con el modelo de caja flexible; simplemente tenemos que crear un contenedor flexible, declarar el tamaño de los elementos que queremos que sean flexibles con la propiedad **`flex`** y aquellos que queremos que tengan un tamaño fijo con la propiedad **`width`** (ver Capítulo 4, Listado 4-37). Pero el modelo de caja tradicional no se diseñó para realizar esta clase de tareas. Por suerte existen algunos

trucos que podemos implementar para lograr este propósito. La alternativa más popular es la de declarar un relleno en la columna flexible con el que hacemos sitio para ubicar la columna de tamaño fijo y agregar un margen negativo que desplaza esta columna al espacio vacío dejado por el relleno. Las siguientes son las reglas que necesitamos para declarar un ancho flexible para el elemento **<section>** y un ancho fijo para el elemento **<aside>** de nuestro documento.

```
* {
  margin: 0px;
  padding: 0px;
}
section {
  float: left;
  width: 100%;
  padding-right: 260px;
  margin-right: -240px;
  box-sizing: border-box;
}
section > div {
  padding: 20px;
  background-color: #999999;
}
aside {
  float: left;
  width: 240px;
  padding: 20px;
  background-color: #CCCCCC;
  box-sizing: border-box;
}
.recuperar {
  clear: both;
}
```

Listado 5-10: *Declarando columnas fijas y flexibles*

Las reglas del Listado 5-10 asignan un ancho de 100 % y un relleno del lado derecho de 260 píxeles al elemento **<section>**. Como usamos la propiedad **box-sizing** para incluir el relleno en el valor de la propiedad **width**, el contenido del elemento ocupará solo el lado izquierdo, dejando un espacio vacío a la derecha donde podemos ubicar la columna con tamaño fijo. Finalmente, para mover la columna creada por el elemento **<aside>** a este espacio, declaramos un margen negativo de 240 píxeles en el lado derecho del elemento **<section>**.

Lo básico: en este ejemplo hacemos flotar ambas columnas a la izquierda, lo que significa que se van a ubicar una al lado de la otra en la misma línea. En casos como estos, podemos crear un espacio en medio de las columnas incrementando el valor de la propiedad **padding**. En el ejemplo del Listado 5-10 declaramos un relleno de 260 píxeles y un margen de 240 píxeles, lo que significa que la columna de tamaño fijo solo se moverá 240 píxeles hacia la izquierda, dejando un espacio entre medio de 20 píxeles.

Debido a que estamos usando el relleno para generar un espacio vacío en el que ubicar la columna de la derecha, tenemos que asignar el relleno normal de la columna y el color de

fondo al elemento contenedor `<div>` dentro del elemento `<section>`. Esto no solo nos permite agregar un relleno al contenido de la columna, sino además asignar un color de fondo solo al área ocupada por el contenido, diferenciando las dos columnas en la pantalla.

***Figura 5-3:** Columnas flexibles y fijas*

Hágalo usted mismo: reemplace las reglas en su archivo adaptabletodos.css por el código del Listado 5-10. Abra el documento en su navegador. Arrastre un lado de la ventana para cambiar su tamaño. Debería ver la columna izquierda expandirse o encogerse, y la columna derecha siempre del mismo tamaño (240 píxeles).

Si queremos ubicar la columna de tamaño fijo a la izquierda en lugar de a la derecha, el proceso es el mismo, pero tenemos que declarar la columna izquierda debajo de la columna derecha en el código (como se han declarado en el Listado 5-6) y luego configurar sus posiciones con la propiedad `float`, tal como hacemos en el siguiente ejemplo.

```
* {
  margin: 0px;
  padding: 0px;
}
section {
  float: right;
  width: 100%;
  padding-left: 260px;
  margin-left: -240px;
  box-sizing: border-box;
}
section > div {
  padding: 20px;
  background-color: #999999;
}
aside {
  float: left;
  width: 240px;
  padding: 20px;
  background-color: #CCCCCC;
  box-sizing: border-box;
}
```

```
.recuperar {
  clear: both;
}
```

Listado 5-11: *Moviendo la columna fija a la izquierda*

El elemento **<section>** se declara primero en nuestro documento, pero debido a que asignamos el valor **right** a su propiedad **float**, se muestra del lado derecho de la pantalla. La posición del elemento en el código asegura que se va a presentar sobre el elemento **<aside>**, y el valor de la propiedad **float** lo mueve al lado que queremos en la página. El resto del código es similar al del ejemplo anterior, excepto que esta vez tenemos que modificar el relleno y el margen del elemento **<section>** del lado izquierdo en lugar del derecho.

Figura 5-4: *Columna fija del lado izquierdo*

Lo básico: cuando movemos un elemento a una nueva posición asignando un margen negativo al siguiente elemento, el navegador presenta el primer elemento detrás del segundo y, por lo tanto, el primer elemento no recibe ningún evento, como los clics del usuario en los enlaces. Para asegurarnos de que la columna de tamaño fijo permanece visible y accesible al usuario, tenemos que declararla en el código debajo de la columna flexible. Esta es la razón por la que para mover la columna creada por el elemento **<aside>** a la izquierda no hemos tenido que modificar el documento. La columna se ubica en su lugar dentro de la página por medio de la propiedad **float**.

Si lo que queremos es declarar dos columnas con tamaños fijos a los lados y una columna flexible en el medio usando este mismo mecanismo, tenemos que agrupar las columnas dentro de un contenedor y luego aplicar las propiedades a los elementos dos veces, dentro y fuera del contenedor. En el siguiente documento, agrupamos dos elementos **<section>** dentro de un elemento **<div>** identificado con el nombre **contenedor** para contener las columnas izquierda y central. Estas dos columnas se han identificado con los nombres **columnaizquierda** y **columnacentral**.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
```

```
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <div id="contenedor">
    <section id="columnacentral">
      <div>
        <article>
          <h1>Título Primer Artículo</h1>
          <img src="miimagen.jpg">
        </article>
      </div>
    </section>
    <section id="columnaizquierda">
      <div>
        <h1>Opciones</h1>
        <p>Opción 1</p>
        <p>Opción 2</p>
      </div>
    </section>
    <div class="recuperar"></div>
  </div>
  <aside>
    <div>
      <h1>Información</h1>
      <p>Cita del artículo uno</p>
      <p>Cita del artículo dos</p>
    </div>
  </aside>
</body>
</html>
```

Listado 5-12: *Creando un documento con dos columnas fijas*

El código CSS para este documento es sencillo. Los elementos **contenedor** y **columnacentral** tienen que declararse como flexibles, por lo que tenemos que asignar al contenedor el relleno y el margen adecuados para ubicar el elemento **<aside>** a la derecha, y luego definir el relleno y el margen del elemento **columnacentral** para hacer sitio para la columna fija del lado izquierdo.

```
* {
  margin: 0px;
  padding: 0px;
}
#contenedor {
  float: left;
  width: 100%;
  padding-right: 260px;
  margin-right: -240px;
  box-sizing: border-box;
}
```

```
aside {
  float: left;
  width: 240px;
  padding: 20px;
  background-color: #CCCCCC;
  box-sizing: border-box;
}
#columnacentral {
  float: right;
  width: 100%;
  padding-left: 220px;
  margin-left: -200px;
  box-sizing: border-box;
}
#columnacentral > div {
  padding: 20px;
  background-color: #999999;
}
#columnaizquierda {
  float: left;
  width: 200px;
  padding: 20px;
  background-color: #CCCCCC;
  box-sizing: border-box;
}
.recuperar {
  clear: both;
}
```

Listado 5-13: *Definiendo dos columnas fijas a los lados*

En este ejemplo declaramos el elemento **columnaizquierda** después del elemento **columnacentral**. Con esto nos aseguramos de que **columnaizquierda** se presenta sobre **columnacentral** y, por lo tanto, es accesible. Como hemos hecho antes, las posiciones en la página web de estos elementos se definen con la propiedad **float**. En consecuencia, el elemento **columnaizquierda** se coloca en el lado izquierdo de la página con un ancho fijo de 200 píxeles, el elemento **columnacentral** se coloca en el centro con un ancho flexible y el elemento **<aside>** a la derecha con un ancho fijo de 240 píxeles.

Figura 5-5: *Columnas fijas a ambos lados*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 5-12 y un archivo CSS llamado misestilos.css con el código del Listado 5-13. Incluya el archivo miimagen.jpg en el mismo directorio. Abra el documento en su navegador y cambie el tamaño de la ventana. Debería ver algo similar a la Figura 5-5, con las columnas a los lados siempre del mismo tamaño.

Texto

Otro aspecto del diseño que tenemos que adaptar a diferentes dispositivos es el texto. Cuando declaramos un tipo de letra con un tamaño fijo, como 14 píxeles, el texto se muestra siempre en ese tamaño, independientemente del dispositivo. En ordenadores personales puede verse bien, pero en pantallas pequeñas un texto de este tamaño será difícil de leer. Para ajustar el tamaño de la letra al dispositivo, podemos usar un tamaño de letra estándar. Este es un tamaño determinado por el navegador de acuerdo con el dispositivo en el que se ejecuta y ajustado a las preferencias del usuario. Por ejemplo, en ordenadores personales, el tamaño de la fuente por defecto es normalmente 16 píxeles. Para asegurarnos de que el texto de nuestra página es legible, podemos definir su tamaño en relación a este valor. CSS ofrece las siguientes unidades de medida con este propósito.

em—Esta unidad representa una medida relativa al tamaño de la fuente asignado al elemento. Acepta números decimales. El valor 1 es igual al tamaño actual de la fuente.

rem—Esta unidad representa una medida relativa al tamaño de la fuente asignada el elemento raíz (usualmente, el elemento `<body>`). Acepta números decimales. El valor 1 es igual al tamaño actual de la fuente.

Las fuentes se definen del mismo modo que lo hemos hecho antes, pero la unidad `px` se debe reemplazar por la unidad `em` para declarar el tamaño de la fuente relativo al tamaño de fuente actual asignado al elemento. Por ejemplo, la siguiente hoja de estilo define tamaños de letra para los elementos `<h1>` y `<p>` en nuestro documento usando unidades `em` (las reglas se han definido considerando el documento del Listado 5-6).

```
* {
  margin: 0px;
  padding: 0px;
}
section {
  float: left;
  width: 61%;
  padding: 2%;
  margin-right: 5%;
  background-color: #999999;
}
aside {
  float: left;
  width: 26%;
  padding: 2%;
  background-color: #CCCCCC;
}
.recuperar {
  clear: both;
}
```

```
article h1 {
  font: bold 2em Georgia, sans-serif;
}
aside h1 {
  font: bold 1.2em Georgia, sans-serif;
}
aside p {
  font: 1em Georgia, sans-serif;
}
```

***Listado 5-14:** Declarando tamaños de letra relativos*

La unidad **em** reemplaza a la unidad **px** y, por lo tanto, el navegador es responsable de calcular el tamaño del texto en píxeles a partir del tamaño de la fuente actualmente asignada al elemento. Para calcular este valor, el navegador multiplica el número de **em** por el tamaño actual de la fuente estándar. Por ejemplo, la regla **article h1** del Listado 5-14 asigna un tamaño de **2em** el elemento **<h1>** dentro de los elementos **<article>**. Esto significa que en un ordenador de escritorio el texto en estos elementos tendrá un tamaño de 32 píxeles (16 * 2).

***Figura 5-6:** Tamaño de letra relativo*

Hágalo usted mismo: el ejemplo del Listado 5-14 asume que estamos trabajando con el documento del Listado 5-6. Reemplace las reglas en el archivo adaptabletodos.css creado para este documento con el código del Listado 5-14 y abra el documento en su navegador. Debería ver algo similar a la Figura 5-6.

La unidad **em** define el tamaño relativo al tamaño de fuente asignado al elemento. Esto se debe a que los elementos heredan propiedades de sus contenedores y, por lo tanto, sus tamaños de letra pueden ser diferentes del estándar establecido por el navegador. Esto significa que si modificamos el tamaño de letra del contenedor de un elemento, el tamaño de letra asignado al elemento se verá afectado. Por ejemplo, las siguiente reglas asignan un tamaño de letra al elemento **<article>** y otro al elemento **<h1>** en su interior.

```
article {
  font: bold 1.5em Georgia, sans-serif;
}
article h1 {
```

```
  font: bold 2em Georgia, sans-serif;
}
```

Listado 5-15: *Declarando tamaños relativos para los elementos y sus contenedores*

La primera regla asigna un tamaño de letra de **`1.5em`** al elemento **`<article>`**. Después de aplicarse esta propiedad, el contenido del elemento **`<article>`**, incluido el contenido del elemento **`<h1>`** en su interior, tendrá un tamaño de 1.5 veces el tamaño estándar (24 píxeles en un ordenador personal). El elemento **`<h1>`** hereda este valor y, por lo tanto, cuando se aplica la siguiente regla, el tamaño de la letra no se calcula desde el tamaño estándar sino desde el tamaño establecido por el contenedor (16 * 1.5 * 2 = 48). El resultado se muestra en la Figura 5-7.

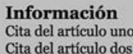

Figura 5-7: *Tamaño de letra relativo al del contenedor*

Si queremos cambiar este comportamiento y forzar al navegador a calcular el tamaño de letra siempre desde el valor estándar, podemos usar las unidades **`rem`**. Estas unidades son relativas al elemento raíz en lugar del elemento actual y, por lo tanto, los valores siempre se multiplican por el mismo valor base.

```
article {
  font: bold 1.5rem Georgia, sans-serif;
}
article h1 {
  font: bold 2rem Georgia, sans-serif;
}
```

Listado 5-16: *Declarando tamaños relativos para los elementos con unidades* `rem`

Si aplicamos las reglas del Listado 5-16 al ejemplo del Listado 5-14, el navegador calcula el tamaño de la fuente a partir del valor estándar y muestra el título del artículo al doble de este tamaño, según ilustra la Figura 5-6, sin importar el tamaño declarado para el contenedor.

Lo básico: también puede usar las unidades **`em`** en lugar de porcentajes para definir el tamaño de los elementos. Cuando utiliza las unidades **`em`**, el tamaño queda determinado por el tamaño de la fuente en lugar del contenedor. Esta técnica se usa para crear diseños elásticos. Para más información, visite nuestro sitio web y siga los enlaces de este capítulo.

Imágenes

Las imágenes se muestran por defecto en su tamaño original, lo que significa que se adaptan al espacio disponible a menos que lo declaremos de forma explícita. Para convertir una imagen fija en una imagen flexible tenemos que declarar su tamaño en porcentaje. Por ejemplo, la siguiente regla configura el ancho de los elementos **<img>** dentro de los elementos **<article>** de nuestro documento con un valor de 100 % y, por lo tanto, las imágenes tendrán un ancho igual al de su contenedor.

```
article img {
  width: 100%;
}
```

Listado 5-17: *Adaptando el tamaño de las imágenes*

Hágalo usted mismo: agregue la regla del Listado 5-17 a la hoja de estilo creada para el ejemplo anterior y abra el documento en su navegador. Debería ver la imagen expandirse o encogerse junto con la columna a medida que cambia el tamaño de la ventana.

Al asignar un porcentaje al ancho de la imagen, se fuerza al navegador a calcular el tamaño de la imagen de acuerdo con el tamaño de su contenedor (el elemento **<article>** en nuestro documento). También podemos declarar valores menores a 100 %, pero el tamaño de la imagen siempre será proporcional al tamaño de su contenedor, lo cual significa que si el contenedor se expande, la imagen se podría presentar con un tamaño más grande que el original. Si queremos establecer los límites para expandir o encoger la imagen, tenemos que usar las propiedades **max-width** y **min-width**. Ya hemos visto estas propiedades aplicadas en el modelo de caja flexible, pero también se pueden emplear en el modelo de caja tradicional para declarar límites para elementos flexibles. Por ejemplo, si queremos que la imagen se reduzca solo cuando no hay espacio suficiente para mostrarla en su tamaño original, podemos usar la propiedad **max-width**.

```
article img {
  max-width: 100%;
}
```

Listado 5-18: *Declarando un tamaño máximo para las imágenes*

La regla del Listado 5-18 deja que la imagen se expanda hasta que alcanza su tamaño original. La imagen será tan ancha como su contenedor a menos que el contenedor tenga un tamaño mayor al tamaño original de la imagen.

Hágalo usted mismo: reemplace la regla del Listado 5-17 en su hoja de estilo con la regla del Listado 5-18. Abra el documento en su navegador y modifique el tamaño de la ventana. Debería ver la imagen expandirse hasta que alcance su tamaño original, tal como muestra la Figura 5-8.

Figura 5-8: *Imagen con un ancho máximo*

Además de adaptar la imagen al espacio disponible, un sitio web adaptable también necesita que unas imágenes se reemplacen por otras cuando las condiciones cambian demasiado. Existen al menos dos situaciones en las que esto es necesario: cuando el espacio disponible no es suficiente para mostrar la imagen original (cuando el logo del sitio web tiene que ser reemplazado por una versión reducida, por ejemplo) o cuando la densidad de píxeles de la pantalla es diferente y necesitamos mostrar una imagen con una resolución más alta o más baja. Independientemente del motivo, HTML ofrece el siguiente elemento para seleccionar la imagen a mostrar.

<picture>—Este elemento es un contenedor que nos permite especificar múltiples fuentes para el mismo elemento **`<img>`**.

<source>—Este elemento define una posible fuente para el elemento **`<img>`**. Puede incluir el atributo **`media`**, para especificar la Media Query a la que la imagen estará asociada, y el atributo **`srcset`** para especificar la ruta de las imágenes que queremos declarar como fuentes del elemento.

El elemento **`<picture>`** es solo un contenedor que debe incluir uno o más elementos **`<source>`** para especificar las posibles fuentes de la imagen y un elemento **`<img>`** al final para mostrar la imagen seleccionada en pantalla. El siguiente documento ilustra cómo reemplazar el logo de un sitio web por una versión simplificada en dispositivos con pantallas pequeñas usando estos elementos.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>
  <header>
    <picture>
      <source media="(max-width: 480px)" srcset="logoreducido.jpg">
      <img src="logo.jpg">
    </picture>
  </header>
```

```
</body>
</html>
```

Listado 5-19: *Seleccionando la imagen con el elemento* `<picture>`

En el documento del Listado 5-19 declaramos solo una fuente para la imagen y la asociamos a la Media Query **`max-width: 480px`**. Cuando el navegador lee este documento, primero procesa el elemento **`<source>`** y luego asigna la fuente al elemento **`<img>`** si el área de visualización cumple los requisitos de la Media Query. Si no se encuentran coincidencias, se muestra en su lugar la imagen especificada por el atributo **`src`** del elemento **`<img>`**. Esto significa que cada vez que el área de visualización es superior a 480 píxeles, la imagen logo.jpg se carga y se muestra en pantalla.

Figura 5-9: *Logo original*

Si el ancho del área de visualización es igual o menor que 480 píxeles, el navegador asigna la imagen especificada por el elemento **`<source>`** al elemento **`<img>`,** y la imagen logoreducido.jpg se muestra en pantalla.

Figura 5-10: *Logo para pantallas pequeñas*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 5-19. Descargue las imágenes logo.jpg y logoreducido.jpg desde nuestro sitio web y muévalas al directorio donde se encuentra el documento. Abra el documento en su navegador y cambie el tamaño de la ventana. Dependiendo del tamaño actual, debería ver algo similar a la Figura 5-9 o la Figura 5-10.

Aunque este es probablemente el escenario más común en el que una imagen debe ser reemplazada por otra de diferente tamaño o contenido, existe una condición más que debemos considerar cuando mostramos imágenes a nuestros usuarios. Desde que se introdujeron los monitores de tipo retina por parte de Apple en el año 2010, existen pantallas con una densidad más alta de píxeles. Una densidad más alta significa más píxeles por pulgada, lo que se traduce en imágenes más claras y definidas. Pero si queremos aprovechar esta característica, debemos facilitar imágenes de alta resolución en estas pantallas. Por ejemplo, una imagen de 300 píxeles de ancho por 200 píxeles de alto dentro de un área del mismo tamaño se verá bien en pantallas con una densidad normal, pero los dispositivos con alta densidad de píxeles tendrán que estirar la imagen para ocupar todos los píxeles disponibles en la misma área. Si queremos que esta imagen también se vea bien en pantallas retina con una escala de 2 (doble cantidad de píxeles por área), tenemos que reemplazarla por la misma imagen con una resolución dos veces superior (600 x 400 píxeles).

Las Media Queries contemplan esta situación con la palabra clave **resolution**. Esta palabra clave requiere que el valor se declare con un número entero y la unidad **dppx**. El valor 1 representa una resolución estándar. Valores más altos definen la escala de la pantalla. Actualmente, las pantallas retina tienen escalas de 2 y 3. El siguiente documento declara una fuente específica para pantallas con una escala de 2 (las más comunes).

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <section>
    <div>
      <article>
        <h1>Título Primer Artículo</h1>
        <picture>
          <source media="(resolution: 2dppx)" srcset="miimagen@2x.jpg">
          <img src="miimagen.jpg">
        </picture>
      </article>
    </div>
  </section>
  <aside>
    <div>
      <h1>Información</h1>
      <p>Cita del artículo uno</p>
      <p>Cita del artículo dos</p>
    </div>
  </aside>
  <div class="recuperar"></div>
</body>
</html>
```

Listado 5-20: *Seleccionando imágenes para diferentes resoluciones*

Si este documento se abre en una pantalla con una densidad de píxeles estándar, el navegador muestra la imagen miimagen.jpg, pero cuando el dispositivo tiene una pantalla retina con una escala de 2, el navegador asigna la imagen miimagen@2x.jpg al elemento **<img>** y la muestra en pantalla.

Como ilustra el ejemplo del Listado 5-20, el proceso es sencillo. Tenemos que crear dos imágenes, una con una resolución normal y otra con una resolución más alta para dispositivos con pantalla retina, y luego declarar las fuentes disponibles con los elementos **<source>**. Pero esto presenta un problema. Si no especificamos el tamaño de las imágenes, estas se muestran en sus tamaños originales y, por lo tanto, el tamaño de la imagen de alta resolución será el doble del de la imagen normal. Por esta razón, cada vez que trabajamos con imágenes de baja y alta resolución, tenemos que declarar sus tamaños de forma explícita. Por ejemplo,

las siguientes reglas configuran el ancho de la imagen con un valor de 100 % del tamaño de su contenedor, lo cual hará que ambas imágenes adopten el mismo tamaño.

```
* {
  margin: 0px;
  padding: 0px;
}
section {
  float: left;
  width: 61%;
  padding: 2%;
  margin-right: 5%;
  background-color: #999999;
}
aside {
  float: left;
  width: 26%;
  padding: 2%;
  background-color: #CCCCCC;
}
.recuperar {
  clear: both;
}
article img {
  width: 100%;
}
```

Listado 5-21: *Declarando el tamaño de una imagen de alta resolución*

Esto es lo mismo que hemos hecho con anterioridad, pero esta vez el navegador muestra imágenes de diferente resolución dependiendo de las características del dispositivo. La Figura 5-11 muestra cómo se ve el documento en una pantalla retina (hemos agregado el texto "2X" en la parte inferior del archivo miimagen@2x.jpg para diferenciar la imagen de baja resolución de la de alta resolución).

Figura 5-11: *Imagen de alta resolución mostrada en una pantalla Retina*

Como hemos hecho en el ejemplo anterior, podemos usar la propiedad `max-width` para limitar el ancho de la imagen a su tamaño original (ver Listado 5-18), pero como esta vez

estamos usando dos imágenes de diferentes tamaños, no podemos declarar el valor de esta propiedad en porcentaje. En su lugar, tenemos que usar el ancho exacto que queremos que sea el máximo permitido. En nuestro ejemplo, la imagen en el archivo miimagen.jpg tiene un ancho original de 365 píxeles.

```
article img {
  width: 100%;
  max-width: 365px;
}
```

Listado 5-22: *Declarando el máximo tamaño de una imagen de alta resolución*

Ahora, sin importar la imagen que selecciona el navegador, se mostrará con un tamaño máximo de 365 píxeles.

Figura 5-12: *Limitaciones para imágenes de alta resolución*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 5-20 y un archivo CSS llamado misestilos.css con el código del Listado 5-21. Descargue las imágenes miimagen.jpg y miimagen@2x.jpg desde nuestro sitio web y muévalas al directorio del documento. Abra el documento en su navegador. Si su ordenador incluye una pantalla Retina, debería ver la imagen con el texto "2X" en la parte inferior, tal como ilustra la Figura 5-11. Actualice la hoja de estilo con la regla del Listado 5-22 y cargue de nuevo la página. Debería ver la imagen expandirse hasta que alcanza su tamaño original (365 píxeles).

Lo básico: existe una convención que sugiere declarar los nombres de los archivos que contienen las imágenes de alta resolución con el mismo nombre que la imagen estándar seguido del carácter @, el valor de la escala, y la letra x, como en @2x o @3x. Esta es la razón por la que asignamos el nombre miimagen@2x.jpg al archivo que contiene la imagen de alta resolución.

Al igual que las imágenes introducidas con el elemento **`<img>`**, las imágenes de fondo también se pueden reemplazar, pero en este caso no necesitamos ningún elemento HTML porque podemos llevar a cabo todo el proceso desde CSS usando las propiedades **`background`** o **`background-image`**. Por este motivo, la imagen no se define en el

documento, sino en la hoja de estilo. El documento solo tiene que incluir el elemento al cual le asignaremos la imagen de fondo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <meta charset="utf-8">
  <meta name="description" content="Este es un documento HTML5">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <header>
    <div id="logo"></div>
  </header>
</body>
</html>
```

Listado 5-23: *Incluyendo el elemento necesario para mostrar la imagen de fondo*

La cabecera de este documento contiene un elemento **<div>** identificado con el nombre **logo** que vamos a usar para mostrar el logo de nuestro sitio web. Como hemos hecho anteriormente, para cambiar el valor de una propiedad de acuerdo a diferentes condiciones, tenemos que definir el valor por defecto y luego declarar Media Queries para modificarlo cuando cambian las circunstancias.

```
* {
  margin: 0px;
  padding: 0px;
}
#logo {
  width: 275px;
  height: 60px;
  background: url("logo.jpg") no-repeat;
}
@media (max-width: 480px) {
  #logo {
    width: 60px;
    background: url("logoreducido.jpg") no-repeat;
  }
}
```

Listado 5-24: *Actualizando la imagen de fondo*

La hoja de estilo del Listado 5-24 define el tamaño del elemento **<div>** igual al tamaño de la imagen que queremos mostrar y luego asigna la imagen logo.jpg como su imagen de fondo (ver Figura 5-9). Si el área de visualización es de 480 píxeles o inferior, la Media Query ajusta el tamaño del elemento **<div>** para que coincida con el tamaño del logo pequeño y asigna la imagen logoreducido.jpg al fondo del elemento (ver Figura 5-10). El efecto que produce este código es exactamente el mismo que el que se logra con el elemento **<picture>** en el

documento del Listado 5-19, excepto que esta vez la imagen no la presenta el elemento **<img>**, sino que se asigna al fondo del elemento **<div>**.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 5-23 y un archivo CSS llamado misestilos.css con el código del Listado 5-24. Descargue las imágenes logo.jpg y logoreducido.jpg desde nuestro sitio web y muévalas al directorio de su documento. Abra el documento en su navegador y modifique el tamaño de la ventana. Dependiendo del tamaño actual, debería ver algo similar a la Figura 5-9 o la Figura 5-10.

Aplicación de la vida real

Crear un sitio web adaptable exige combinar todas las técnicas que hemos estudiado. Algunas se deben aplicar varias veces y, normalmente, se debe declarar más de una Media Query para adaptar el diseño a múltiples dispositivos. Pero el sitio en el que establecemos los puntos de interrupción y cómo se implementan estas técnicas depende del diseño de nuestro sitio web y del modelo de caja que implementemos. Para demostrar cómo crear un sitio web con diseño web adaptable y el modelo de caja tradicional, vamos a usar el documento del Capítulo 4, Listado 4-20. La estructura básica de este documento incluye una cabecera, una barra de navegación, dos columnas creadas con los elementos **<section>** y **<aside>**, y un pie de página. Lo primero es definir los estilos por defecto que se necesitan para transformar estos elementos en elementos flexibles.

Como hemos mencionado anteriormente, para volver a un elemento flexible en el modelo de caja tradicional, tenemos que definir sus anchos con valores en porcentaje. Los siguientes son los estilos por defecto requeridos por el documento del Capítulo 4, Listado 4-20 (estos son los estilos que se aplicarán si ninguna Media Query coincide con el tamaño del área de visualización). A diferencia de lo que hemos hecho en el Capítulo 4, esta vez los tamaños se declaran en porcentaje y usamos la propiedad **max-width** para especificar un ancho máximo para el contenido de la página.

```
* {
  margin: 0px;
  padding: 0px;
}
#cabeceralogo {
  width: 96%;
  height: 150px;
  padding: 0% 2%;
  background-color: #0F76A0;
}
#cabeceralogo > div {
  max-width: 960px;
  margin: 0px auto;
  padding-top: 45px;
}
#cabeceralogo h1 {
  font: bold 54px Arial, sans-serif;
  color: #FFFFFF;
}
#menuprincipal {
  width: 96%;
  height: 50px;
```

```
  padding: 0% 2%;
  background-color: #9FC8D9;
  border-top: 1px solid #094660;
  border-bottom: 1px solid #094660;
}
#menuprincipal > div {
  max-width: 960px;
  margin: 0px auto;
}
#menuprincipal li {
  display: inline-block;
  height: 35px;
  padding: 15px 10px 0px 10px;
  margin-right: 5px;
}
#menuprincipal li:hover {
  background-color: #6FACC6;
}
#menuprincipal a {
  font: bold 18px Arial, sans-serif;
  color: #333333;
  text-decoration: none;
}
main {
  width: 96%;
  padding: 2%;
  background-image: url("fondo.png");
}
main > div {
  max-width: 960px;
  margin: 0px auto;
}
#articulosprincipales {
  float: left;
  width: 65%;
  padding-top: 30px;
  background-color: #FFFFFF;
  border-radius: 10px;
}
#infoadicional {
  float: right;
  width: 29%;
  padding: 2%;
  background-color: #E7F1F5;
  border-radius: 10px;
}
#infoadicional h1 {
  font: bold 18px Arial, sans-serif;
  color: #333333;
  margin-bottom: 15px;
}
.recuperar {
  clear: both;
}
article {
  position: relative;
```

```
  padding: 0px 40px 20px 40px;
}
article time {
  display: block;
  position: absolute;
  top: -5px;
  left: -70px;
  width: 80px;
  padding: 15px 5px;
  background-color: #094660;
  box-shadow: 3px 3px 5px rgba(100, 100, 100, 0.7);
  border-radius: 5px;
}
.numerodia {
  font: bold 36px Verdana, sans-serif;
  color: #FFFFFF;
  text-align: center;
}
.nombredia {
  font: 12px Verdana, sans-serif;
  color: #FFFFFF;
  text-align: center;
}
article h1 {
  margin-bottom: 5px;
  font: bold 30px Georgia, sans-serif;
}
article p {
  font: 18px Georgia, sans-serif;
}
figure {
  margin: 10px 0px;
}
figure img {
  max-width: 98%;
  padding: 1%;
  border: 1px solid;
}
#pielogo {
  width: 96%;
  padding: 2%;
  background-color: #0F76A0;
}
#pielogo > div {
  max-width: 960px;
  margin: 0px auto;
  background-color: #9FC8D9;
  border-radius: 10px;
}
.seccionpie {
  float: left;
  width: 27.33%;
  padding: 3%;
}
.seccionpie h1 {
  font: bold 20px Arial, sans-serif;
}
```

```
.seccionpie p {
  margin-top: 5px;
}
.seccionpie a {
  font: bold 16px Arial, sans-serif;
  color: #666666;
  text-decoration: none;
}
```

Listado 5-25: *Definiendo elementos flexibles con el modelo de caja tradicional*

Las reglas del Listado 5-25 vuelven flexibles los elementos estructurales, de modo que el sitio web se adapta al espacio disponible, pero cuando la pantalla es demasiado pequeña, el diseño se rompe, algunos elementos se muestran de forma parcial y el contenido se vuelve imposible de leer, tal como muestra la Figura 5-13.

Figura 5-13: *Sitio web adaptable solo con elementos flexibles*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 4-20. Cree un archivo CSS llamado misestilos.css e incluya las reglas del Listado 5-25. Abra el documento en su navegador. A este momento, los elementos se expanden o encogen de acuerdo con el espacio disponible, pero la página no se ve bien cuando el área de visualización es muy pequeña (ver Figura 5-13). Agregaremos Media Queries a la hoja de estilo a continuación para corregir esta situación.

Los cambios en el diseño se tienen que introducir gradualmente. Por ejemplo, en nuestro diseño, cuando el tamaño del área de visualización es de 1120 píxeles o inferior, el elemento **<time>** que usamos para representar la fecha en la que el artículo se ha publicado se queda fuera de la ventana. Esto nos indica que nuestro diseño necesita un punto de interrupción a 1120 píxeles para mover este elemento a una posición diferente o reorganizar el contenido. En este caso, decidimos mover la fecha de vuelta dentro del área ocupada por el elemento **<article>**.

```
@media (max-width: 1120px) {
  article time {
    position: static;
    width: 100%;
```

```
    padding: 0px;
    margin-bottom: 10px;

    background-color: #FFFFFF;
    box-shadow: 0px 0px 0px;
    border-radius: 0px;
  }
  .numerodia {
    display: inline-block;
    font: bold 14px Verdana, sans-serif;
    color: #999999;
    padding-right: 5px;
  }
  .nombredia {
    display: inline-block;
    font: bold 14px Verdana, sans-serif;
    color: #999999;
  }
  article h1 {
    margin-bottom: 0px;
  }
}
```

Listado 5-26: *Reposicionando el elemento* `<time>`

Lo primero que tenemos que hacer para reposicionar la fecha es restaurar el modo de posicionamiento del elemento **`<time>`**. Las reglas por defecto introducidas en la hoja de estilo del Listado 5-25 le otorgan una posición absoluta al elemento para moverlo al lado derecho del área ocupada por el elemento **`<article>`**, pero cuando la pantalla no es lo suficientemente grande para mostrarlo en esa posición, tenemos que moverlo nuevamente a su ubicación natural en el documento, debajo del elemento **`<h1>`**. Esto se logra asignando el valor **`static`** a la propiedad **`position`** (**`static`** es el modo por defecto). Ahora, el elemento **`<time>`** se coloca debajo del título del artículo, pero aún tenemos que ubicar la fecha y el día en la misma línea. Para este ejemplo, decidimos convertir los elementos en elementos **`inline-block`**, por lo que se ubicarán uno al lado del otro en la misma línea (Figura 5-14).

Figura 5-14: *La fecha se desplaza a la posición debajo del título con Media Queries*

Hágalo usted mismo: agregue la Media Query definida en el Listado 5-26 al final de su hoja de estilo y actualice la página en su navegador. Reduzca el tamaño de la ventana para ver cómo se modifica el elemento **`<time>`**.

Otro momento en el que se debe modificar el diseño es cuando las dos columnas se vuelven demasiado pequeñas para mostrar el contenido apropiadamente. Dependiendo de las características del contenido, podemos optar por ocultarlo, reemplazarlo con un contenido diferente o reorganizar las columnas. En este caso, decidimos convertir el diseño de dos columnas en un diseño de una columna moviendo el elemento **<aside>** debajo del elemento **<section>**. Existen varias maneras de lograr este objetivo. Por ejemplo, podemos asignar el valor **none** a la propiedad **float** para prevenir que los elementos floten a los lados, o simplemente definir el ancho de los elemento con los valores 100 % o **auto**, y dejar que el navegador se encargue de ubicarlos uno por línea. Para nuestro ejemplo, decidimos establecer un diseño de una sola columna cuando el área de visualización tiene un ancho de 720 píxeles o menos usando la segunda opción.

```
@media (max-width: 720px) {
  #articulosprincipales {
    width: 100%;
  }
  #infoadicional {
    width: 90%;
    padding: 5%;
    margin-top: 20px;
  }
}
```

Listado 5-27: *Reorganizando las columnas*

Cada vez que el documento se muestre en una pantalla de un tamaño de 720 píxeles o inferior, el usuario verá el contenido organizado en una sola columna.

Figura 5-15: *Diseño de una columna*

Hágalo usted mismo: agregue la Media Query definida en el Listado 5-27 al final de su hoja de estilo y actualice la página en su navegador. Cuando el ancho del área de visualización sea de 720 píxeles o inferior, debería ver el contenido en una sola columna, tal como ilustra la Figura 5-15.

En este momento, el documento se ve bien en ordenadores personales y tablets, pero aún debemos realizar varios cambios para adaptarlo a las pequeñas pantallas de los teléfonos móviles. Cuando esta página se muestre en un área de visualización de 480 píxeles o inferior, las opciones del menú no tendrán espacio suficiente para visualizarse en una sola línea, y el

contenido del pie de página puede que no tenga espacio suficiente para mostrarse por completo. Una manera de resolver este problema es listando los ítems uno encima del otro.

```
@media (max-width: 480px) {
  #cabeceralogo > div {
    text-align: center;
  }
  #cabeceralogo h1 {
    font: bold 46px Arial, sans-serif;
  }
  #menuprincipal {
    width: 100%;
    height: 100%;
    padding: 0%;
  }
  #menuprincipal li {
    display: block;
    margin-right: 0px;
    text-align: center;
  }
  .seccionpie {
    width: 94%;
    text-align: center;
  }
}
```

Listado 5-28: *Creando un menú móvil*

Con las reglas del Listado 5-28 modificamos los elementos para forzar al navegador a mostrarlos uno por línea y centrar sus contenidos. Las primeras dos reglas centran el contenido del elemento **cabeceralogo** y asignan un nuevo tamaño al título de la página para que se muestre mejor en pantallas pequeñas. A continuación, definimos el tamaño del elemento **menuprincipal** (el contenedor del menú) para que tenga el máximo ancho posible y una altura determinada por su contenido (**height: 100%**). También declaramos los elementos **<li>** dentro del elemento **menuprincipal** como elementos Block para mostrar las opciones del menú una por línea. Finalmente, los anchos de las tres secciones dentro del pie de página también se extienden para forzar al navegador a mostrarlas una por línea. La Figura 5-16 ilustra cómo afectan estos cambios a algunos de los elementos.

Figura 5-16: *Menú móvil*

Hágalo usted mismo: agregue la Media Query definida en el Listado 5-28 al final de su hoja de estilo y actualice la página en su navegador. Reduzca el tamaño de la ventana para ver cómo se adaptan al espacio disponible las opciones del menú y las secciones del pie de página.

Después de aplicar estas reglas, el pie de página se ve bien, pero las opciones del menú de la parte superior de la pantalla desplazan el contenido relevante hacia abajo, forzando al usuario a desplazar la página para poder verlo. Una alternativa es reemplazar el menú con un botón y mostrar las opciones solo cuando se pulsa el botón. Para este propósito, tenemos que agregar una imagen al documento que ocupará el lugar del menú cuando el ancho del área de visualización sea de 480 píxeles o inferior.

```
<nav id="menuicono">
  <img src="iconomenu.png" width="35" height="28" id="menu-img">
</nav>
<nav id="menuprincipal">
  <div>
    <ul>
      <li><a href="index.html">Principal</a></li>
      <li><a href="fotos.html">Fotos</a></li>
      <li><a href="videos.html">Videos</a></li>
      <li><a href="contacto.html">Contacto</a></li>
    </ul>
  </div>
</nav>
```

Listado 5-29: *Agregando el botón del menú para pantallas pequeñas*

La primera modificación que tenemos que introducir en nuestra hoja de estilo es una regla que oculta el elemento `menuicono` porque solo lo queremos mostrar en pantallas pequeñas. Existen dos formas de hacerlo: definir la propiedad `visibility` con el valor `hidden` o declarar el modo de visualización como `none` con la propiedad `display`. La primera opción no muestra el elemento al usuario, pero el elemento aún ocupa un espacio en la página, mientras que la segunda le dice al navegador que debe mostrar la página como si el elemento se hubiera incluido en el documento y, por lo tanto, esta última es la opción que tenemos que implementar para nuestro menú.

```
#menuicono {
  display: none;
  width: 95%;
  height: 38px;
  padding: 12px 2% 0px 3%;
  background-color: #9FC8D9;
  border-top: 1px solid #094660;
  border-bottom: 1px solid #094660;
}
```

Listado 5-30: *Ocultando el botón del menú*

El siguiente paso es mostrar el botón y ocultar el menú cuando el ancho del área de visualización es de 480 píxeles o inferior. Las siguientes son las modificaciones que tenemos que introducir en este punto de interrupción.

```
@media (max-width: 480px) {
  #menuprincipal {
    display: none;
    width: 100%;
    height: 100%;
    padding: 0%;
  }
  #menuprincipal li {
    display: block;
    margin-right: 0px;
    text-align: center;
  }
  #menuicono {
    display: block;
  }
  .seccionpie {
    width: 94%;
    text-align: center;
  }
  #cabeceralogo > div {
    text-align: center;
  }
}
```

Listado 5-31: *Reemplazando el menú con el botón*

Asignando el valor **none** a la propiedad **display** del elemento **menuprincipal** hacemos que el menú desaparezca. Si el ancho del área de visualización es de 480 píxeles o inferior, el elemento **menuicono** y su contenido se muestran en su lugar.

Figura 5-17: *Botón del menú*

Hágalo usted mismo: agregue un elemento **<nav>** identificado con el nombre **menuicono** encima del elemento **<nav>** que ya existe en su documento, como muestra el Listado 5-29. Agregue la regla del Listado 5-30 en la parte superior de su hoja de estilo. Actualice la Media Query para el punto de interrupción 480px con el código del Listado 5-31. Abra el documento en su navegador y reduzca el tamaño de la ventana. Debería ver el nuevo botón ocupando el lugar del menú cuando el ancho del área de visualización es de 480 píxeles o inferior.

En este momento, tenemos un menú que se adapta al espacio disponible, pero el botón no responde. Para mostrar el menú cuando el usuario pulsa o hace clic en el botón, tenemos que agregar a nuestro documento un programa que responda a esta acción. Estas acciones se llaman *eventos* y son controladas por código escrito en JavaScript. Estudiaremos JavaScript y eventos en el Capítulo 6, pero la tarea que debemos realizar aquí es sencilla. Tenemos que volver visible al elemento **menuprincipal** cuando el usuario pulsa el botón. La siguiente es nuestra implementación para este ejemplo.

```
<script>
  var visible = false;
  function iniciar() {
    var elemento = document.getElementById("menu-img");
    elemento.addEventListener("click", mostrarMenu);
  }
  function mostrarMenu() {
    var elemento = document.getElementById("menuprincipal");
    if (!visible) {
      elemento.style.display = "block";
      visible = true;
    } else {
      elemento.style.display = "none";
      visible = false;
    }
  }
  window.addEventListener("load", iniciar);
</script>
```

Listado 5-32: *Mostrando el menú cuando se pulsa el botón*

Como veremos más adelante, una forma de insertar código JavaScript dentro de un documento HTML es por medio del elemento **<script>**. Este elemento se ubica normalmente dentro de la cabecera (el elemento **<head>**), pero también se puede ubicar en cualquier otra parte del documento.

El código JavaScript, como cualquier otro código de programación, está compuesto por una serie de instrucciones que se procesan de forma secuencial. El código del Listado 5-32 primero obtiene una referencia al elemento **menu-img** y agrega una función que responderá al evento **click** de este elemento. Luego, cuando el elemento recibe el clic del usuario, el código cambia el valor de la propiedad **display** del elemento **menuprincipal** dependiendo de la condición actual. Si el menú no es visible, lo hace visible, o viceversa (explicaremos cómo funciona este código en el Capítulo 6). La Figura 5-18 muestra lo que vemos cuando se pulsa el botón.

Hágalo usted mismo: agregue el código del Listado 5-32 dentro del elemento **<head>** y debajo del elemento **<link>** en su documento. Actualice la página en su navegador. Haga clic en el botón para abrir el menú. Debería ver algo similar a la Figura 5-18.

Hasta el momento, hemos trabajado con el modelo de caja tradicional. Aunque este es el modelo preferido hoy en día debido a su compatibilidad con las versiones antiguas de los navegadores, también tenemos la opción de implementar el diseño web adaptable con el modelo de caja flexible. Para demostrar cómo desarrollar un sitio web adaptable con este modelo, vamos a usar el documento del Capítulo 4, Listado 4-52.

Figura 5-18: *Menú mostrado por código JavaScript*

Este ejemplo asume que hemos incluido en el documento el elemento **<nav>** agregado en el Listado 5-29 y el elemento **<script>** con el código JavaScript introducido en el Listado 5-32, pero este código se tiene que modificar para que trabaje con este modelo. El valor asignado a la propiedad **display** para hacer que el menú aparezca cuando el usuario pulsa el botón tiene que ser **flex** en lugar de **block,** porque ahora estamos trabajando con contenedores flexibles.

```
<script>
  var visible = false;
  function iniciar() {
    var elemento = document.getElementById("menu-img");
    elemento.addEventListener("click", mostrarMenu);
  }
  function mostrarMenu() {
    var elemento = document.getElementById("menuprincipal");
    if (!visible) {
      elemento.style.display = "flex";
      visible = true;
    } else {
      elemento.style.display = "none";
      visible = false;
    }
  }
  window.addEventListener("load", iniciar);
</script>
```

Listado 5-33: *Mostrando el menú como un contenedor flexible*

La hoja de estilo que necesitamos es muy parecida a la que usamos con el modelo de caja tradicional; la única diferencia es que tenemos que construir contenedores flexibles y declarar los elementos flexibles con la propiedad **flex**. Los siguientes son los estilos por defecto para todo el documento.

```
* {
  margin: 0px;
  padding: 0px;
}
#cabeceralogo {
  display: flex;
```

```
  justify-content: center;
  width: 96%;
  height: 150px;
  padding: 0% 2%;
  background-color: #0F76A0;
}
#cabeceralogo > div {
  flex: 1;
  max-width: 960px;
  padding-top: 45px;
}
#cabeceralogo h1 {
  font: bold 54px Arial, sans-serif;
  color: #FFFFFF;
}
#menuprincipal {
  display: flex;
  justify-content: center;
  width: 96%;
  height: 50px;
  padding: 0% 2%;
  background-color: #9FC8D9;
  border-top: 1px solid #094660;
  border-bottom: 1px solid #094660;
}
#menuprincipal > div {
  flex: 1;
  max-width: 960px;
}
#menuprincipal li {
  display: inline-block;
  height: 35px;
  padding: 15px 10px 0px 10px;
  margin-right: 5px;
}
#menuprincipal li:hover {
  background-color: #6FACC6;
}
#menuprincipal a {
  font: bold 18px Arial, sans-serif;
  color: #333333;
  text-decoration: none;
}
#menuicono {
  display: none;
  width: 95%;
  height: 38px;
  padding: 12px 2% 0px 3%;
  background-color: #9FC8D9;
  border-top: 1px solid #094660;
  border-bottom: 1px solid #094660;
}
main {
  display: flex;
  justify-content: center;
  width: 96%;
```

```
  padding: 2%;
  background-image: url("fondo.png");
}
main > div {
  display: flex;
  flex: 1;
  max-width: 960px;
}
#articulosprincipales {
  flex: 1;
  margin-right: 20px;
  padding-top: 30px;
  background-color: #FFFFFF;
  border-radius: 10px;
}
#infoadicional {
  flex: 1;
  max-width: 280px;
  padding: 2%;
  background-color: #E7F1F5;
  border-radius: 10px;
}
#infoadicional h1 {
  font: bold 18px Arial, sans-serif;
  color: #333333;
  margin-bottom: 15px;
}
article {
  position: relative;
  padding: 0px 40px 20px 40px;
}
article time {
  display: block;
  position: absolute;
  top: -5px;
  left: -70px;
  width: 80px;
  padding: 15px 5px;
  background-color: #094660;
  box-shadow: 3px 3px 5px rgba(100, 100, 100, 0.7);
  border-radius: 5px;
}
.numerodia {
  font: bold 36px Verdana, sans-serif;
  color: #FFFFFF;
  text-align: center;
}
.nombredia {
  font: 12px Verdana, sans-serif;
  color: #FFFFFF;
  text-align: center;
}
article h1 {
  margin-bottom: 5px;
  font: bold 30px Georgia, sans-serif;
}
```

```
article p {
  font: 18px Georgia, sans-serif;
}
figure {
  margin: 10px 0px;
}
figure img {
  max-width: 98%;
  padding: 1%;
  border: 1px solid;
}
#pielogo {
  display: flex;
  justify-content: center;
  width: 96%;
  padding: 2%;
  background-color: #0F76A0;
}
#pielogo > div {
  display: flex;
  flex: 1;
  max-width: 960px;
  background-color: #9FC8D9;
  border-radius: 10px;
}
.seccionpie {
  flex: 1;
  padding: 3%;
}
.seccionpie h1 {
  font: bold 20px Arial, sans-serif;
}
.seccionpie p {
  margin-top: 5px;
}
.seccionpie a {
  font: bold 16px Arial, sans-serif;
  color: #666666;
  text-decoration: none;
}
```

Listado 5-34: *Diseñando un documento adaptable con el modelo de caja flexible*

El diseño gráfico para este documento es el mismo que creamos anteriormente, por lo que debemos establecer los mismos puntos de interrupción. Nuevamente, cuando el ancho del área de visualización es de 1120 píxeles o inferior, tenemos que mover el elemento **<time>** debajo del título del artículo. Debido a que en ambos modelos el elemento **<time>** se posiciona con valores absolutos, esta Media Query no presenta cambio alguno.

```
@media (max-width: 1120px) {
  article time {
    position: static;
    width: 100%;
    padding: 0px;
```

```
    margin-bottom: 10px;
    background-color: #FFFFFF;
    box-shadow: 0px 0px 0px;
    border-radius: 0px;
  }
  .numerodia {
    display: inline-block;
    font: bold 14px Verdana, sans-serif;
    color: #999999;
    padding-right: 5px;
  }
  .nombredia {
    display: inline-block;
    font: bold 14px Verdana, sans-serif;
    color: #999999;
  }
  article h1 {
    margin-bottom: 0px;
  }
}
```

Listado 5-35: *Moviendo el elemento* `<time>`

El paso siguiente es convertir el diseño de dos columnas en un diseño de una columna cuando el ancho del área de visualización es de 720 píxeles o inferior. Debido a que ya no queremos que las columnas compartan la misma línea, sino que se muestren una encima de la otra, tenemos que declarar el contenedor como un elemento Block. Una vez que lo hemos hecho, es fácil extender los elementos hacia los lados, solo tenemos que darles un tamaño de 100 % (debido a que el elemento **`<aside>`** tiene por defecto un ancho máximo de 280 píxeles, también tenemos que declarar el valor de la propiedad **`max-width`** como 100 % para eliminar esta limitación).

```
@media (max-width: 720px) {
  main > div {
    display: block;
  }
  #articulosprincipales {
    width: 100%;
    margin-right: 0px;
  }
  #infoadicional {
    width: 90%;
    max-width: 100%;
    padding: 5%;
    margin-top: 20px;
  }
}
```

Listado 5-36: *Pasando de un diseño de dos columnas a un diseño de una columna*

En el último punto de interrupción tenemos que modificar la barra del menú para mostrar el botón del menú en lugar de las opciones y declarar el contenedor en el pie de página como un elemento Block para ubicar una sección sobre la otra.

```
@media (max-width: 480px) {
  #cabeceralogo > div {
    text-align: center;
  }
  #cabeceralogo h1 {
    font: bold 46px Arial, sans-serif;
  }
  #menuprincipal {
    display: none;
    width: 100%;
    height: 100%;
    padding: 0%;
  }
  #menuprincipal li {
    display: block;
    margin-right: 0px;
    text-align: center;
  }
  #menuicono {
    display: block;
  }
  #pielogo > div {
    display: block;
  }
  .seccionpie {
    width: 94%;
    text-align: center;
  }
}
```

Listado 5-37: *Adaptando el menú y el pie de página*

Con el código del Listado 5-37, la hoja de estilo está lista. El diseño final es exactamente igual que el que logramos con el modelo de caja tradicional, pero esta vez usando el modelo de caja flexible. El modelo de caja flexible es una gran mejora con respecto al modelo de caja tradicional y puede simplificar la creación de sitios web adaptables, permitiéndonos modificar el orden en el que se presentan los elementos y facilitando la combinación de elementos de tamaño flexible y fijos, aunque no es compatible con todos los navegadores del mercado. Algunos desarrolladores ya utilizan este modelo o implementan algunas de sus propiedades, pero la mayoría de los sitios web aún se desarrollan con el modelo de caja tradicional.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 4-52 (vea el modelo de caja flexible en el Capítulo 4). Agregue el elemento **`<nav>`** introducido en el Listado 5-29 y el elemento **`<script>`** del Listado 5-33 al documento tal como hemos explicado en el ejemplo anterior. Cree un nuevo archivo CSS llamado misestilos.css y copie los códigos de los Listados 5-34, 5-35, 5-36 y 5-37 en su interior. Descargue los archivos miimagen.jpg e iconomenu.png desde nuestro sitio web y muévalos al directorio de su documento. Abra el documento en su navegador y cambie el tamaño de la ventana para ver cómo se adaptan los elementos al espacio disponible.

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!».

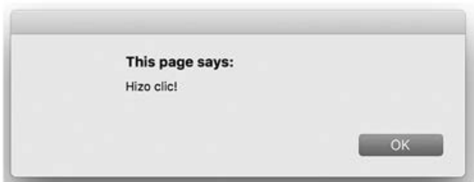

Figura 6-1: *Ventana emergente generada por la función* `alert()`

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).

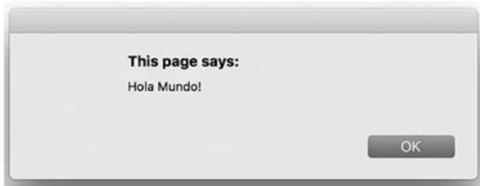

Figura 6-3: *Ventana emergente mostrando una cadena de caracteres*

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*

El ejemplo del Listado 6-51 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.

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 6-52: *Declarando variables globales y locales*

El código del Listado 6-52 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 6-53: *Declarando dos variables con el mismo nombre*

En el código del Listado 6-53, 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 6-54: *Enviando un valor a una función*

En el ejemplo del Listado 6-54, 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 6-55: *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 6-56: *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 6-57: *Enviando múltiples valores a la función*

En el ejemplo del Listado 6-57, 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 6-58: *Devolviendo valores desde funciones*

El código del Listado 6-58 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 6-59: *Devolviendo diferentes valores desde una función*

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 6-60: *Declarando funciones anónimas*

En el ejemplo del Listado 6-60, 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 6-61: *Ejecutando funciones anónimas*

La función del Listado 6-61 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.

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 6-62: *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 6-63: *Convirtiendo una cadena de caracteres en un número*

El código del Listado 6-63 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 6-64: *Codificando una cadena de caracteres para incluirla en una URL*

El código del Listado 6-64 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.

6.3 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.

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 6-65: *Creando objetos*

En el ejemplo del Listado 6-65 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 6-66: *Accediendo propiedades*

En el Listado 6-66, 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 6-67: *Accediendo propiedades usando variables*

En el Listado 6-67, 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 6-68: *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 6-69: *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 6-70: *Creando objetos dentro de objetos*

El objeto **`miobjeto`** en el código del Listado 6-70 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`**.

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 6-71: *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 6-72: *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.

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 6-73: *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 6-73 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.

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 6-74: *Usando un constructor para crear un objeto*

En el ejemplo del Listado 6-74 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 6-75: *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 6-76: *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 6-76, 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 6-77: *Definiendo propiedades privadas*

El código del Listado 6-77 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)**).

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 6-78: *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 6-79: *Definiendo valores iniciales para el objeto*

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 6-80: *Creando objetos a partir de otros objetos*

El código del Listado 6-80 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 6-81: *Agregando un nuevo método al prototipo*

En el Listado 6-81, 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 6-82: *Probando la cadena de prototipos*

En el Listado 6-82, 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.

6.4 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 6-83: *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 6-84: *Creando números a partir de cadenas de caracteres*

La cadena de caracteres que provee al constructor `Number()` en el Listado 6-84 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 6-85: *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 6-86: *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 6-87: *Creando un array vacío con un constructor*

Lo básico: el método que utilicemos para asignar un valor depende de nosotros. Podemos asignar los valores directamente como hemos hecho en ejemplos previos o usar estos constructores. El resultado es el mismo. JavaScript es capaz de determinar la diferencia entre un valor y un objeto, pero generalmente esto resulta irrelevante, y la mayoría del tiempo los dos tipos de valores se comportan de la misma manera.

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 en la cadena.

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.

283áxim(separador, limite**)**—Este método divide la cadena de caracteres y devuelve un array con todas las partes. El atributo **separador** indica el carácter en el que se va a cortar la cadena y el atributo **limite** es un número entero que determina el número máximo de partes. Si no especificamos un límite, la cadena se cortará cada vez que se encuentre el separador.

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 6-88: *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 6-89: *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 6-89 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 6-90: *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 6-91: *Procesando una cadena de caracteres*

El código del Listado 6-91 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 6-92: *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 6-93: *Copiando un grupo de caracteres*

El método **substr()** del Listado 6-93 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 6-94: *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 6-95: *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 6-96: *Copiando caracteres entre dos índices*

El método **substring()** del Listado 6-96 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 **287áxim()**. 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 6-97: *Obteniendo las palabras de una cadena*

El método **287áxim()** del Listado 6-97 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 6-98: *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 6-98, 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 6-99: *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 6-100: *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 6-101: *Reemplazando textos en una cadena de caracteres*

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 6-102: *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 6-89). 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 6-103: *Iterando sobre el array*

En el código del Listado 6-103, 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 6-104: *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 6-104, 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 6-105: *Filtrando los valores de un array*

En la función provista para el método **filter()** del Listado 6-105, 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 6-106: *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 6-107: *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 6-108: *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 del array 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 6-109: *Obteniendo el índice de un valor en el array*

El código del Listado 6-109 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 6-110: *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 6-110, 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 6-111: *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 6-112: *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 6-113: *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 6-114: *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 6-115: *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 6-116: *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 6-117: *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 6-118: *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 6-119: *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 6-120: *Procesando los valores y almacenando los resultados*

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 6-121: *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 6-122: *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 6-123: *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 6-124: *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 6-125: *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 6-126: *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 6-127: *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 6-128: *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 6-129: *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 6-130: *Comparando dos fechas*

El código del Listado 6-130 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 6-131: *Comparando solo las fechas sin considerar la hora*

En el Listado 6-131, 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.

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 devuelve el 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 valores separados 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 logaritmo base 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 6-132: *Ejecutando operaciones aritméticas con el objeto* `Math`

El código del Listado 6-132 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 6-133: *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()**.

Lo básico: los métodos **floor()** y **ceil()** se usan ampliamente en aplicaciones JavaScript. El método **floor()** redondea un número hacia abajo al entero más cercano y el método **ceil()** redondea el número hacia arriba al entero más cercano. Por ejemplo, si el número a ser procesado es 5.86, el método **floor()** devuelve el valor 5 y el método **ceil()** devuelve el valor 6. Implementaremos estos métodos y otros del objeto **Math** en situaciones prácticas en próximos capítulos.

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 6-134: *Definiendo una nueva ubicación*

En el Listado 6-134 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.

Hágalo usted mismo: cree un nuevo archivo HTML con el Listado 6-134 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 6-135: *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 6-136: *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 6-137: *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 6-138:** Cancelando un temporizador*

El código del Listado 6-138 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.

Hágalo usted mismo: reemplace el documento en su archivo HTML con el código del Listado 6-138 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.

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 6-139: *Obteniendo una referencia al objeto* `Element` *que representa un elemento*

El código del Listado 6-139 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 6-140: *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 6-141: *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 6-141, 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 6-142: *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 6-143: *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 6-144: *Buscando un elemento dentro de otro elemento*

El código del Listado 6-144 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.

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 6-145: *Asignando nuevos estilos al documento*

El código del Listado 6-145 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 6-4.

Figura 6-4: *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 6-146: *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 6-5: *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 6-147: *Reemplazando la clase del elemento*

En el código del Listado 6-147 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 6-148: *Activando y desactivando clases*

Con el código del Listado 6-148, 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 6-149:** 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.

Hágalo usted mismo: 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 6-150: *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 6-151: *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 6-152: *Insertando código HTML en el documento*

El código del Listado 6-152 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 6-153: *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 6-153, 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 6-6.

Figura 6-6: *Contenido agregado a un elemento*

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 6-154: *Creando objetos* `Element`

El código del Listado 6-154 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 6-155: *Agregando contenido a un objeto* `Element`

Figura 6-7: *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.

6.5 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 6-156: *Definiendo atributos de eventos desde código JavaScript*

El documento del Listado 6-156 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.

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 6-157: *Respondiendo a eventos con el método* addEventListener()

El código del Listado 6-157 es el mismo que el del ejemplo anterior, pero ahora utilizamos el método **addEventListener()** para agregar listeners al objeto **Window** y el elemento **<button>**.

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 6-158: *Usando objetos* `Event`

El código del Listado 6-158 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 6-8: *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 6-159: *Respondiendo a un evento con una función anónima*

El código del Listado 6-159 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.

Sitio Web

Mensaje número 1

Valor 125

Mensaje número 3

Figura 6-9: *El elemento se modifica con los valores recibidos por la función*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 6-158. 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 6-159 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 6-9.

En el ejemplo del Listado 6-159, 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 6-160: *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 6-10: *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 6-161: *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.

Sitio Web

Valor 0

Valor 1

Valor 2

Figura 6-11: *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 6-162: *Información la posición del ratón*

El código del Listado 6-162 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 6-163: *Calculando la posición del ratón en un elemento*

El documento del Listado 6-163 incluye dos elementos **`<div>`**, uno dentro del otro, para recrear una barra de progreso. El elemento **`<div>`** 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 6-12: *Barra de progreso*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 6-163 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 6-12.

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 6-164: *Detectando la tecla pulsada*

El documento del Listado 6-164 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 6-165: *Detectando teclas especiales*

Como hemos hecho en el ejemplo del Listado 6-146, 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.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 6-165. 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.

6.6 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.

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 6-13: *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 6-166: *Generando un error*

En el Listado 6-166 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.

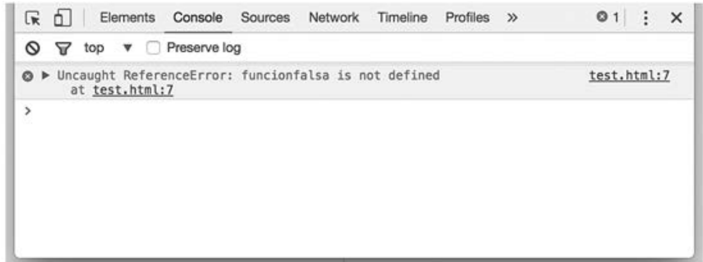

Figura 6-14: *Error reportado en la consola*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 6-166 y abra el documento en su navegador. Acceda al menú principal del 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 6-14.

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 6-167: *Mostrando mensajes en la consola con el método* `log()`

El código del Listado 6-167 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.

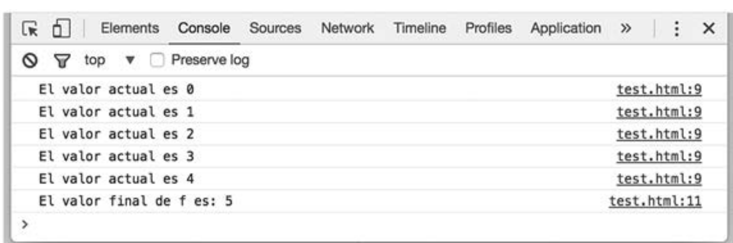

Figura 6-15: *Mensajes en la consola*

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.

Lo básico: el método `log()` también puede imprimir objetos en la consola, lo que nos permite leer el contenido de un objeto desconocido e identificar sus propiedades y métodos.

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 6-168: Respondiendo a errores

En el código del Listado 6-168, 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`**.

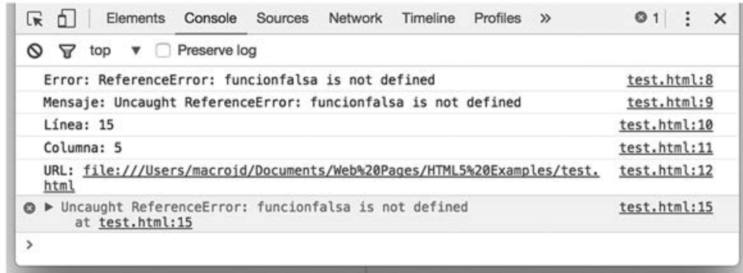

Figura 6-16: *Información acerca del error*

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 6-169: 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".

6.7 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.

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.

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 6-170: *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 6-170.

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".

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 6-170. 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.

Capítulo 7
API Formularios

7.1 Procesando formularios

La API Formularios es un grupo de propiedades, métodos y eventos que podemos usar para procesar formularios y crear nuestro propio sistema de validación. La API está integrada en los formularios y elementos de formularios, por lo que podemos responder a eventos o llamar a los métodos desde los mismos elementos. Los siguientes son algunos de los métodos disponibles para el elemento **<form>**.

submit()—Este método envía el formulario.

reset()—Este método reinicializa el formulario (asigna los valores por defecto a los elementos).

checkValidity()—Este método devuelve un valor booleano que indica si el formulario es válido o no.

La API también ofrece el siguiente evento para anunciar cada vez que se inserta un carácter o se selecciona un valor en un elemento de formulario.

input—Este evento se desencadena en el formulario o sus elementos cuando el usuario inserta o elimina un carácter en un campo de entrada, y también cuando se selecciona un nuevo valor.

change—Este evento se desencadena en el formulario o sus elementos cuando se introduce o selecciona un nuevo valor.

Usando estos métodos y eventos, podemos controlar el proceso de envío del formulario desde JavaScript. El siguiente ejemplo envía el formulario cuando se pulsa un botón.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
  <script>
    function iniciar() {
      var boton = document.getElementById("enviar");
      boton.addEventListener("click", enviarformulario);
    }
    function enviarformulario() {
      var formulario = document.querySelector("form[name='informacion']");
      formulario.submit();
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
```

```
<body>
  <section>
    <form name="informacion" method="get" action="procesar.php">
      <p><label>Correo: <input type="email" name="correo" id="correo"
required></label></p>
      <p><button type="button" id="enviar">Registrarse</button></p>
    </form>
  </section>
</body>
</html>
```

Listado 7-1: *Enviando un formulario desde JavaScript*

El código del Listado 7-1 responde al evento **`click`** desde el elemento **`<button>`** para ejecutar la función **`enviarformulario()`** cada vez que se pulsa el botón. En esta función, obtenemos una referencia al elemento **`<form>`** y luego enviamos el formulario con el método **`submit()`**.

En este ejemplo, hemos decidido obtener la referencia al elemento **`<form>`** con el método **`querySelector()`** y un selector que busca el elemento por medio de su atributo **`name`**, pero podríamos haber agregado un atributo **`id`** al elemento para obtener la referencia con el método **`getElementById()`**, como hemos hecho con el botón. Otra alternativa es obtener la referencia desde la propiedad **`forms`** del objeto **`Document`**. Esta propiedad devuelve un array con referencias a todos los elementos **`<form>`** del documento. En nuestro caso, solo tenemos un elemento **`<form>`** y, por lo tanto, la referencia se encuentra en el índice 0.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
  <script>
    function iniciar() {
      var boton = document.getElementById("enviar");
      boton.addEventListener("click", enviarformulario); }
    function enviarformulario() {
      var lista = document.forms;
      var formulario = lista[0];
      formulario.submit(); }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <form name="informacion" method="get" action="procesar.php">
      <p><label>Correo: <input type="email" name="correo" id="correo"
required></label></p>
      <p><button type="button" id="enviar">Registrarse</button></p>
    </form>
  </section>
</body>
</html>
```

Listado 7-2: *Obteniendo una referencia al elemento* `<form>` *desde la propiedad* `forms`

Enviar el formulario con el método **submit()** es lo mismo que hacerlo con un elemento **<input>** de tipo **submit** (ver Capítulo 2), la diferencia es que este método evita el proceso de validación del navegador. Si queremos que el formulario sea validado, tenemos que hacerlo nosotros con el método **checkValidity()**. Este método controla el formulario y devuelve **true** o **false** para indicar si es válido o no.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
  <script>
    function iniciar() {
      var boton = document.getElementById("enviar");
      boton.addEventListener("click", enviarformulario);
    }
    function enviarformulario() {
      var formulario = document.querySelector("form[name='informacion']");
      var valido = formulario.checkValidity();
      if (valido) {
        formulario.submit();
      } else {
        alert("El formulario no puede ser enviado");
      }
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <form name="informacion" method="get" action="procesar.php">
      <p><label>Correo: <input type="email" name="correo" id="correo"
required></label></p>
      <p><button type="button" id="enviar">Registrarse</button></p>
    </form>
  </section>
</body>
</html>
```

Listado 7-3: *Controlando la validez del formulario*

El código del Listado 7-3 controla los valores en el formulario para determinar su validez. Si el formulario es válido, se envía con el método **submit()**. En caso contrario, se muestra un mensaje en pantalla para advertir al usuario.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 7-3. Abra el documento en su navegador e intente enviar el formulario. Debería ver una ventana emergente advirtiéndole de que el formulario no se puede enviar. Inserte una cuenta de correo válida en el campo. Ahora, el formulario sí se debería enviar.

7.2 Validación

Como hemos visto en el Capítulo 2, existen diferentes maneras de validar formularios en HTML. Podemos usar campos de entrada del tipo que requieren validación por defecto, como **`email`**, convertir un campo regular de tipo **`text`** en un campo requerido con el atributo **`required`**, o incluso usar tipos especiales como **`pattern`** para personalizar los requisitos de validación. Sin embargo, cuando tenemos que implementar mecanismos de validación más complejos, como comparar dos o más campos o controlar el resultado de una operación, nuestra única opción es la de personalizar el proceso de validación usando la API Formularios.

Errores personalizados

Los navegadores muestran un mensaje de error cuando el usuario intenta enviar un formulario que contiene un campo no válido. Estos son mensajes predefinidos que describen errores conocidos, pero podemos definir mensajes personalizados para establecer nuestros propios requisitos de validación. Con este fin, los objetos **`Element`** que representan elementos de formulario incluyen el siguiente método.

setCustomValidity(mensaje**)**—Este método declara un error personalizado y el mensaje a mostrar si se envía el formulario. Si no se especifica ningún mensaje, el error se anula.

El siguiente ejemplo presenta una situación de validación compleja. Se crean dos campos para recibir el nombre y el apellido del usuario. Sin embargo, el formulario solo es no válido cuando ambos campos están vacíos. El usuario puede introducir su nombre o su apellido para validarlo. En un caso como este, es imposible usar el atributo **`required`** porque no sabemos qué campo va a elegir completar el usuario. Solo usando errores personalizados podemos crear un mecanismo de validación efectivo para este escenario.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
  <script>
    var nombre1, nombre2;
    function iniciar() {
      nombre1 = document.getElementById("nombre");
      nombre2 = document.getElementById("apellido");
      nombre1.addEventListener("input", validacion);
      nombre2.addEventListener("input", validacion);
      validacion();
    }
    function validacion() {
      if (nombre1.value == "" && nombre2.value == "") {
        nombre1.setCustomValidity("Inserte su nombre o su apellido");
        nombre1.style.background = "#FFDDDD";
        nombre2.style.background = "#FFDDDD";
      } else {
        nombre1.setCustomValidity("");
        nombre1.style.background = "#FFFFFF";
```

```
            nombre2.style.background = "#FFFFFF";
          }
        }
        window.addEventListener("load", iniciar);
      </script>
    </head>
    <body>
      <section>
        <form name="registracion" method="get" action="procesar.php">
          <p><label>Nombre: <input type="text" name="nombre"
id="nombre"></label></p>
          <p><label>Apellido: <input type="text" name="apellido"
id="apellido"></label></p>
          <p><input type="submit" value="Registrarse"></p>
        </form>
      </section>
    </body>
    </html>
```

Listado 7-4: *Declarando mensajes de error personalizados*

El código del Listado 7-4 comienza creando referencias a dos elementos **<input>** y agregando listeners al evento **input** para cada uno de ellos. Este evento se desencadena cada vez que el usuario inserta o elimina un carácter, lo que nos permite detectar cada valor insertado en los campos, y validar o invalidar el formulario desde la función **validacion()**.

Debido a que los elementos **<input>** se encuentran vacíos cuando se carga el documento, tenemos que declarar una condición no válida para no permitir al usuario enviar el formulario antes de insertar al menos uno de los valores en los campos. Por esta razón, la función **validacion()** también se llama al final de la función **iniciar()** para comprobar esta condición.

La función **validacion()** controla si el formulario es válido o no, y declara o elimina el error con el método **setCustomValidity()**. Si ambos campos están vacíos, se declara un error personalizado y el color de fondo de ambos elementos se cambia a rojo para indicar al usuario el error. Sin embargo, si la condición cambia debido a que se ha introducido al menos uno de los valores, el error se elimina llamando al método con una cadena de caracteres vacía y el color blanco se asigna nuevamente al fondo de ambos campos.

Es importante recordar que el único cambio producido durante el proceso es la modificación del color de fondo de los campos. El mensaje de error que declara el método **setCustomValidity()** solo se mostrará al usuario cuando intente enviar el formulario.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 7-4 y abra el documento en su navegador. Intente enviar el formulario. Debería ver un error con el mensaje "Inserte su nombre o apellido". Inserte un valor. El error se debería eliminar.

Lo básico: se pueden declarar varias variables separadas por comas en la misma línea. En el Listado 7-4 hemos declarado dos variables globales llamadas **nombre1** y **nombre2**. Esta instrucción no es necesaria porque las variables declaradas dentro de funciones sin el operador **var** se asignan al ámbito global, pero declarar las variables que vamos a utilizar al comienzo del código simplifica su mantenimiento porque nos ayuda a identificar sin demasiado esfuerzo las variables que nuestro código necesita para trabajar.

El evento invalid

Cada vez que el usuario envía un formulario, se desencadena un evento si se detecta un campo no válido. El evento se llama **invalid** y se desencadena en el elemento que ha producido el error. Para personalizar una respuesta, podemos responder a este evento desde el elemento **<form>**, como lo hacemos en el siguiente ejemplo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
  <script>
    var formulario;
    function iniciar() {
      var boton = document.getElementById("enviar");
      boton.addEventListener("click", enviarformulario);
      formulario = document.querySelector("form[name='informacion']");
      formulario.addEventListener("invalid", validacion, true);
    }
    function validacion(evento) {
      var elemento = evento.target;
      elemento.style.background = "#FFDDDD";
    }
    function enviarformulario() {
      var valido = formulario.checkValidity();
      if (valido) {
        formulario.submit();
      }
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <form name="informacion" method="get" action="procesar.php">
      <p><label>Apodo: <input pattern="[A-Za-z]{3,}" name="apodo"
id="apodo" maxlength="10" required></label></p>
      <p><label>Correo: <input type="email" name="correo" id="correo"
required></label></p>
      <p><button type="button" id="enviar">Registrarse</button></p>
    </form>
  </section>
</body>
</html>
```

Listado 7-5: *Creando un sistema de validación personalizado*

En el Listado 7-5 hemos creado un nuevo formulario con dos campos de entrada para introducir un apodo y una cuenta de correo. El campo **correo** tiene sus limitaciones naturales debido a su tipo y un atributo **required** que lo declara como campo obligatorio, pero el campo **apodo** contiene tres atributos de validación: el atributo **pattern** que solo admite un mínimo de tres caracteres de la A a la Z (mayúsculas y minúsculas), el atributo **maxlength** que limita el campo a un máximo de diez caracteres, y el atributo **required** que invalida el campo si está vacío.

El código es muy similar a los ejemplos anteriores. Respondemos al evento **load** con la función **iniciar()** cuando el documento termina de cargarse y al evento **click** del elemento **<button>**, como siempre, pero luego agregamos un listener para el evento **invalid** al elemento **<form>** en lugar de los campos **<input>**. Esto se debe a que queremos establecer un sistema de validación para todo el formulario, no solo elementos individuales. Para este propósito, tenemos que incluir el valor **true** como tercer atributo del método **addEventListener()**. Este atributo le indica al navegador que tiene que propagar el evento al resto de los elementos de la jerarquía. Como resultado, a pesar de que el *listener* se ha agregado al elemento **<form>**, este responde a eventos desencadenados por los elementos dentro del formulario. Para determinar cuál es el elemento no válido que ha llamado a la función **validacion()**, leemos el valor de la propiedad **target**. Como hemos visto en capítulos anteriores, esta propiedad devuelve una referencia al elemento que desencadenó el evento. Usando esta referencia, la última instrucción en esta función cambia el color de fondo del elemento a rojo.

El objeto ValidityState

El documento del Listado 7-5 no realiza una validación en tiempo real. Los campos se validan solo cuando se envía el formulario. Considerando la necesidad de un sistema de validación más dinámico, la API Formularios incluye el objeto **ValidityState**. Este objeto ofrece una serie de propiedades para indicar el estado de validez de un elemento del formulario.

valid—Esta propiedad devuelve **true** si el valor del elemento es válido.

La propiedad **valid** devuelve el estado de validez de un elemento considerando todos los demás estados de validez. Si todas las condiciones son válidas, la propiedad **valid** devuelve **true**. Si queremos controlar una condición en particular, podemos leer el resto de las propiedades que ofrece el objeto **ValidityState**.

valueMissing—Esta propiedad devuelve **true** cuando se ha declarado el atributo **required** y el campo está vacío.

typeMismatch—Esta propiedad devuelve **true** cuando la sintaxis del texto introducido no coincide con el tipo de campo. Por ejemplo, cuando el texto introducido en un campo de tipo **email** no es una cuenta de correo.

patternMismatch—Esta propiedad devuelve **true** cuando el texto introducido no respeta el formato establecido por el atributo **pattern**.

tooLong—Esta propiedad devuelve **true** cuando se ha declarado el atributo **maxlength** y el texto introducido es más largo que el valor especificado por este atributo.

rangeUnderflow—Esta propiedad devuelve **true** cuando se ha declarado el atributo **min** y el valor introducido es menor que el especificado por este atributo.

rangeOverflow—Esta propiedad devuelve **true** cuando se ha declarado el atributo **max** y el valor introducido es mayor que el especificado por este atributo.

stepMismatch—Esta propiedad devuelve **true** cuando se ha declarado el atributo **step** y el valor introducido no corresponde con el valor de los atributos **min**, **max** y **value**.

customError—Esta propiedad devuelve **true** cuando declaramos un error personalizado con el método **setCustomValidity()**.

El objeto **ValidityState** se asigna a una propiedad llamada **validity**, disponible en cada elemento de formulario. El siguiente ejemplo lee el valor de esta propiedad para determinar la validez de los elementos en el formulario dinámicamente.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
  <script>
    var formulario;
    function iniciar() {
      var boton = document.getElementById("enviar");
      boton.addEventListener("click", enviarformulario);
      formulario = document.querySelector("form[name='informacion']");
      formulario.addEventListener("invalid", validacion, true);
      formulario.addEventListener("input", comprobar);
    }
    function validacion(evento) {
      var elemento = evento.target;
      elemento.style.background = "#FFDDDD";
    }
    function enviarformulario() {
      var valido = formulario.checkValidity();
      if (valido) {
        formulario.submit();
      }
    }
    function comprobar(evento) {
      var elemento = evento.target;
      if (elemento.validity.valid) {
        elemento.style.background = "#FFFFFF";
      } else {
        elemento.style.background = "#FFDDDD";
      }
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <form name="informacion" method="get" action="procesar.php">
      <p><label>Apodo: <input pattern="[A-Za-z]{3,}" name="apodo"
id="apodo" maxlength="10" required></label></p>
      <p><label>Correo: <input type="email" name="correo" id="correo"
required></label></p>
      <p><button type="button" id="enviar">Registrarse</button></p>
    </form>
  </section>
</body>
</html>
```

Listado 7-6: *Validación en tiempo real*

En el código del Listado 7-6, agregamos un listener para el evento **`input`** al formulario. Cada vez que el usuario modifica un campo, insertando o eliminando un carácter, se ejecuta la función **`comprobar()`** para responder al evento.

La función **`comprobar()`** también aprovecha la propiedad **`target`** para obtener una referencia al elemento que desencadenó el evento y controlar su validez leyendo el valor de la propiedad **`valid`** dentro de la propiedad **`validity`** del objeto **`Element`** (**`elemento.validity.valid`**). Con esta información, cambiamos el color de fondo del elemento que desencadenó el evento **`input`** en tiempo real. El color será rojo hasta que el texto introducido por el usuario sea válido.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 7-6. Abra el documento en su navegador e inserte valores en los campos de entrada. Debería ver el color de fondo de los campos cambiar según su validez (válido blanco, no válido rojo).

Podemos usar el resto de las propiedades que ofrece el objeto **`ValidityState`** para saber exactamente qué ha producido el error, tal como muestra el siguiente ejemplo.

```
function enviarformulario() {
  var elemento = document.getElementById("apodo");
  var valido = formulario.checkValidity();
  if (valido) {
    formulario.submit();
  } else if (elemento.validity.patternMismatch ||
elemento.validity.valueMissing) {
    alert('El apodo debe tener un mínimo de 3 caracteres');
  }
}
```

Listado 7-7: *Leyendo los estados de validez para mostrar un mensaje de error específico*

En el Listado 7-7 se modifica la función **`enviarformulario()`** para detectar errores específicos. El formulario lo valida el método **`checkValidity()`** y, si es válido, se envía con el método **`submit()`**. En caso contrario, se leen los valores de las propiedades **`patternMismatch`** y **`valueMissing`** del campo **`apodo`** y se muestra un mensaje de error cuando una o ambas devuelven **`true`**.

Hágalo usted mismo: reemplace la función **`enviarformulario()`** en el documento del Listado 7-6 con la nueva función del Listado 7-7 y abra el documento en su navegador. Escriba un solo carácter en el campo **`apodo`** y envíe el formulario. Debería ver una ventana emergente pidiéndole que inserte un mínimo de tres caracteres.

7.3 Seudoclases

Además de todas las propiedades y métodos provistos por la API Formularios, CSS incluye algunas seudoclases para modificar los estilos de un elemento dependiendo de su estado, incluidos inválido, válido, requerido, opcional, e incluso cuando un valor se encuentra fuera del rango permitido.

Valid e Invalid

Estas seudoclases afectan a cualquier elemento **<input>** con un valor válido o inválido.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
  <style>
    input:valid{
      background: #EEEEFF;
    }
    input:invalid{
      background: #FFEEEE;
    }
  </style>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <input type="email" name="correo" required>
      <input type="submit" value="Enviar">
    </form>
  </section>
</body>
</html>
```

Listado 7-8: *Usando las seudoclases* `:valid` *e* `:invalid`

El formulario del Listado 7-8 incluye un elemento **<input>** para cuentas de correo. Cuando el contenido del elemento no es válido, la seudoclase **:valid** asigna un color de fondo azul al campo, pero tan pronto como el contenido se vuelve no válido, la seudoclase **:invalid** cambia el color de fondo a rojo.

Optional y Required

Estas seudoclases afectan a todos los elementos de formulario declarados como obligatorios u opcionales.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
  <style>
    input:optional{
      border: 2px solid #009999;
    }
    input:required{
      border: 2px solid #000099;
    }
```

```
  </style>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <p><input type="text" name="nombre"></p>
      <p><input type="text" name="apellido" required></p>
      <p><input type="submit" value="Enviar"></p>
    </form>
  </section>
</body>
</html>
```

Listado 7-9: *Usando las seudoclases* `:required` *y* `:optional`

El ejemplo del Listado 7-9 incluye dos campos de entrada: **`nombre`** y **`apellido`**. El primero es opcional, pero **`apellido`** es obligatorio. Las seudoclases asignan un color de borde diferente a estos campos de acuerdo con su condición (el campo obligatorio se muestra en azul y el campo opcional en verde).

In-range y Out-of-range

Estas seudoclases afectan a todos los elementos con un valor dentro o fuera de un rango específico.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Formularios</title>
  <style>
    input:in-range{
      background: #EEEEFF;
    }
    input:out-of-range{
      background: #FFEEEE;
    }
  </style>
</head>
<body>
  <section>
    <form name="formulario" method="get" action="procesar.php">
      <input type="number" name="numero" min="0" max="10">
      <input type="submit" value="Enviar">
    </form>
  </section>
</body>
</html>
```

Listado 7-10: *Usando las seudoclases* `:in-range` *y* `:out-of-range`

En este ejemplo se ha incluido un campo de entrada de tipo `number` para probar estas seudoclases. Cuando el valor introducido en el elemento es menor que `0` o mayor que `10`, el color de fondo es rojo, pero tan pronto como introducimos un valor dentro del rango especificado, el color de fondo cambia a azul.

Capítulo 8
Medios

8.1 Vídeo

Los vídeos son un método extremadamente eficaz de comunicación. Nadie puede negar la importancia de los vídeos en los sitios web y aplicaciones de hoy en día, y mucho menos aquellos que se encargan de desarrollar las tecnologías para la Web. Esta es la razón por la que HTML5 incluye un elemento con el único propósito de cargar y reproducir vídeos.

<video>—Este elemento inserta un vídeo en el documento.

El elemento **`<video>`** incluye los siguientes atributos para declarar el área que ocupa el vídeo y configurar el reproductor.

src—Esta atributo especifica la URL del vídeo a reproducir.

width—Este atributo determina el ancho del área del reproductor.

height—Este atributo determina la altura del área del reproductor.

controls—Este es un atributo booleano. Si está presente, el navegador muestra una interfaz para permitir al usuario controlar el vídeo.

autoplay—Este es un atributo booleano. Si está presente, el navegador reproduce el vídeo automáticamente tan pronto como puede.

loop—Este es un atributo booleano. Si está presente, el navegador reproduce el vídeo una y otra vez.

muted—Este es un atributo booleano. Si está presente, el audio se silencia.

poster—Este atributo especifica la URL de la imagen que se mostrará mientras el navegador espera que el vídeo se reproduzca.

preload—Este atributo determina si el navegador debería comenzar a cargar el vídeo antes de ser reproducido. Acepta tres valores: **`none`**, **`metadata`** o **`auto`**. El primer valor indica que el vídeo no se debería cargar y generalmente se utiliza para minimizar tráfico web. El segundo valor, **`metadata`**, recomienda al navegador descargar información acerca del recurso, como las dimensiones, la duración, el primer cuadro, etc. El tercer valor, **`auto`**, solicita al navegador que descargue el archivo tan pronto como sea posible (este es el valor por defecto).

El elemento **`<video>`** requiere etiquetas de apertura y cierre, y solo algunos parámetros para cumplir su función. La sintaxis es sencilla y solo es obligatorio el atributo **`src`**.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
```

```
  <title>Reproductor de Video</title>
</head>
<body>
  <section>
    <video src="trailer.mp4">
    </video>
  </section>
</body>
</html>
```

Listado 8-1: *Cargando un vídeo con el elemento* `<video>`

El elemento **`<video>`** carga el vídeo especificado por el atributo **`src`** y reserva un área del tamaño del vídeo en el documento, pero el vídeo se reproduce. Tenemos que indicarle al navegador cuándo queremos que reproduzca el vídeo o facilitar las herramientas para dejar que el usuario decida. Existen dos atributos que podemos agregar al elemento para este propósito: **`controls`** y **`autoplay`**. El atributo **`controls`** indica al navegador que debería incluir sus propios elementos (botones y barras) para permitir al usuario controlar el vídeo, y el atributo **`autoplay`** solicita al navegador que comience a reproducir el vídeo tan pronto como pueda.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Reproductor de Video</title>
</head>
<body>
  <section>
    <video src="trailer.mp4" controls autoplay>
    </video>
  </section>
</body>
</html>
```

Listado 8-2: *Activando los controles por defecto*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 8-2. Descargue el vídeo trailer.mp4 desde nuestro sitio web. Abra el documento en su navegador. El navegador debería comenzar a reproducir el vídeo de inmediato y facilitar botones para controlarlo.

Por defecto, los navegadores determinan el tamaño de área del vídeo a partir de su tamaño original, pero podemos definir un tamaño personalizado con los atributos **`width`** y **`height`**. Estos atributos son como los atributos del elemento **`<img>`**, declaran las dimensiones del elemento en píxeles. Cuando están presentes, el tamaño del vídeo se ajusta para adaptarlo a estas dimensiones, pero no tienen el propósito de comprimir o expandir el vídeo. Podemos usarlos para limitar el área ocupada por el medio y preservar la coherencia en nuestro diseño.

```
<!DOCTYPE html>
<html lang="es">
```

```
<head>
  <meta charset="utf-8">
  <title>Reproductor de Video</title>
</head>
<body>
  <section>
    <video src="trailer.mp4" width="720" height="400" controls>
    </video>
  </section>
</body>
</html>
```

Listado 8-3: *Definiendo el área del vídeo*

Lo básico: si el vídeo se incorpora en un sitio web con diseño web adaptable, puede ignorar los atributos `width` y `height`, y declarar su tamaño por medio de CSS y media queries. El elemento `<video>` se puede adaptar al tamaño de su contenedor, como en el caso de las imágenes del Capítulo 5.

El elemento `<video>` incluye atributos adicionales que pueden resultar útiles en algunas aplicaciones. Por ejemplo, el atributo `preload` solicita al navegador que comience a descargar el vídeo tan pronto como pueda, de modo que cuando el usuario decide reproducirlo, la reproducción comienza de inmediato. También contamos con el atributo `loop` para reproducir el vídeo una y otra vez, y con el atributo `poster` para especificar una imagen que se mostrará en lugar del vídeo mientras este no se reproduce.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Reproductor de Video</title>
</head>
<body>
  <section>
    <video src="trailer.mp4" width="720" height="400" preload controls
loop poster="poster.jpg">
    </video>
  </section>
</body>
</html>
```

Listado 8-4: *Incluyendo una imagen que represente el vídeo*

El documento del Listado 8-4 carga el vídeo tan pronto como se carga el documento, reproduce el vídeo continuamente y muestra una imagen en lugar del vídeo mientras este no se reproduce. La Figura 8-1 muestra lo que vemos antes de pulsar el botón para iniciar la reproducción.

Hágalo usted mismo: actualice su documento con el código del Listado 8-4. Descargue el archivo poster.jpg desde nuestro sitio web y abra el documento en su navegador. Debería ver algo similar a la Figura 8-1.

Figura 8-1: *Poster*
© Derechos Reservados 2008, Blender Foundation/www.bigbuckbunny.org

Formatos de vídeo

En teoría, el elemento **<video>** por sí solo debería ser más que suficiente para cargar y reproducir un vídeo, pero el proceso es un poco más complicado en la vida real. Esto se debe a que, a pesar de que el elemento **<video>** y sus atributos son estándar, no existe un formato de vídeo estándar para la web. El problema es que algunos navegadores admiten un grupo de codificadores y otros no, y el codificador que se usa en el formato MP4 (el único que admiten navegadores destacados como Safari e Internet Explorer) se distribuye bajo licencia comercial.

Las opciones más comunes actualmente son OGG, MP4, y WebM. Estos formatos son contenedores de vídeo y audio. OGG contiene codificadores de vídeo Theora y audio Vorbis, MP4 contiene H.264 para vídeo y AAC para audio, y WebM usa VP8 para vídeo Vorbis para audio. Actualmente, OGG y WebM son compatibles con Mozilla Firefox, Google Chrome y Opera, mientras que MP4 funciona en Safari, Internet Explorer y Google Chrome.

HTML contempla este escenario e incluye un elemento adicional que funciona con el elemento **<video>** para definir las posibles fuentes del vídeo.

<source>—Este elemento define una fuente para un vídeo. Debe incluir el atributo **src** para indicar la URL del archivo.

Cuando necesitamos especificar múltiples fuentes para el mismo vídeo, tenemos que reemplazar el atributo **src** en el elemento **<video>** por elementos **<source>** entre las etiquetas, como en el siguiente ejemplo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Reproductor de Video</title>
</head>

<body>
  <section>
    <video width="720" height="400" controls>
      <source src="trailer.mp4">
```

```
      <source src="trailer.ogg">
    </video>
  </section>
</body>
</html>
```

Listado 8-5: *Creando un reproductor de vídeo para varios navegadores*

En el Listado 8-5 el elemento **<video>** se expande. Ahora, entre las etiquetas del elemento hay dos elementos **<source>**. Estos elementos facilitan diferentes fuentes de vídeo para que el navegador elija. El navegador lee estos elementos y decide qué archivo se debería reproducir de acuerdo con los formatos que admite (en este caso, MP4 u OGG).

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 8-5. Descargue los archivos trailer.mp4 y trailer.ogg desde nuestro sitio web. Abra el documento en su navegador. El vídeo se debería reproducir como siempre, pero ahora el navegador selecciona qué fuente usar.

IMPORTANTE: los navegadores requieren que los vídeos se envíen en el servidor con los correspondientes tipos MIME. Cada archivo tiene un tipo MIME asociado para indicar el formato de su contenido. Por ejemplo, el tipo MIME para un archivo HTML es **text/html**. Los servidores ya están configurados para la mayoría de los formatos de vídeo, pero normalmente no para nuevos formatos como OGG o WEBM. La forma de incluir este nuevo tipo MIME depende del tipo de servidor. Una manera sencilla de hacerlo es agregar una nueva línea al archivo **.htaccess**. La mayoría de los servidores incluyen este archivo de configuración en el directorio raíz de todo sitio web. La sintaxis correspondiente es **Addtype MIME/type extension** (por ejemplo, **AddType video/ogg ogg**).

8.2 Audio

El audio es un medio que no tiene la misma popularidad en la Web que los vídeos. Podemos filmar un vídeo con una cámara personal que será visto por millones de personas, pero lograr el mismo resultado con un archivo de audio sería casi imposible. Sin embargo, el audio aún se encuentra presente en la Web a través de shows de radio y podcasts. Por esta razón, HTML5 también ofrece un elemento para reproducir archivos de audio.

<audio>—Este elemento inserta audio en el documento.

Este elemento trabaja de la misma forma y comparte varios atributos con el elemento **<video>**.

src—Este atributo especifica la URL del archivo a reproducir.

controls—Este es un atributo booleano. Si está presente, activa la interfaz que facilita el navegador por defecto.

autoplay—Este es un atributo booleano. Si está presente, el navegador reproduce el audio automáticamente tan pronto como puede.

loop—Este es un atributo booleano. Si está presente, el navegador reproduce el audio una y otra vez.

preload—Este atributo determina si el navegador debe comenzar a cargar el archivo de audio antes de reproducirse. Acepta tres valores: **none**, **metadata** o **auto**. El primer valor indica que el audio no se debería cargar y generalmente se utiliza para minimizar tráfico web. El segundo valor, **metadata**, recomienda al navegador que descargue información acerca del recurso, como la duración del audio. El tercer valor, **auto**, solicita al navegador que descargue el archivo tan pronto como le sea posible (este es el valor por defecto).

La implementación del elemento **<audio>** en nuestro documento es muy similar a la del elemento **<video>**. Solo tenemos que especificar la fuente y ofrecer al usuario la posibilidad de iniciar la reproducción.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Reproductor de Audio</title>
</head>
<body>
  <section>
    <audio src="beach.mp3" controls>
    </audio>
  </section>
</body>
</html>
```

Listado 8-6: *Reproduciendo audio con el elemento* <audio>

Nuevamente tenemos que hablar de codificadores, y una vez más debemos decir que el código HTML del Listado 8-6 debería ser suficiente para reproducir un sonido en la Web, pero no lo es. MP3 se encuentra bajo licencia comercial, por lo que no es compatible con navegadores como Mozilla Firefox u Opera. Vorbis (el codificador de audio del contenedor OGG) es compatible con estos navegadores, pero no con Safari e Internet Explorer. Por lo tanto, una vez más, tenemos que usar el elemento **<source>** para facilitar al menos dos formatos para que el navegador pueda elegir.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Reproductor de Audio</title>
</head>
<body>
  <section>
    <audio id="medio" controls>
      <source src="beach.mp3">
      <source src="beach.ogg">
    </audio>
  </section>
```

```
</body>
</html>
```

Listado 8-7: *Creando un reproductor de audio para varios navegadores*

El documento del Listado 8-7 reproduce una canción en todos los navegadores con controles por defecto. Aquellos que no pueden reproducir MP3 usarán el archivo OGG, y viceversa. Solo tenemos que recordar que MP3, así como el formato MP4 para vídeo, se distribuyen bajo licencias comerciales y solo podemos emplearlos bajo las circunstancias permitidas por sus licencias.

8.3 API Media

Si agregamos el atributo **controls** a los elementos **<video>** y **<audio>** activamos la interfaz por defecto que facilita cada navegador. Esta interfaz puede resultar útil en sitios web sencillos o pequeñas aplicaciones, pero en un ambiente profesional, donde cada detalle cuenta, es necesario disponer de un control absoluto sobre todo el proceso y facilitar un diseño coherente para todos los dispositivos y aplicaciones. Los navegadores incluyen la API Media para ofrecer una alternativa personalizada. Esta API es un grupo de propiedades, métodos y eventos diseñados para manipular e integrar vídeo y audio en nuestro documentos. Combinando esta API con HTML y CSS podemos crear nuestros propios reproductores de vídeo o audio con los controles que queramos.

La API se incluye en los objetos **Element** que representan los elementos **<video>** y **<audio>**. Las siguientes son algunas de las propiedades que incluyen estos objetos para facilitar información acerca del medio.

paused—Esta propiedad devuelve **true** si el medio se ha pausado o aún no ha comenzado a reproducirse.

ended—Esta propiedad devuelve **true** si el medio ha terminado de reproducirse.

duration—Esta propiedad devuelve la duración del medio en segundos.

currentTime—Esta propiedad declara o devuelve un valor que determina la posición en la que el medio se está reproduciendo o debería comenzar a reproducirse.

volume—Esta propiedad declara o devuelve el volumen del medio. Acepta valores entre 0.0 y 1.0.

muted—Esta propiedad declara o devuelve el estado del audio. Los valores son **true** (silenciado) o **false** (no silenciado).

error—Esta propiedad devuelve el valor del error ocurrido.

buffered—Esta propiedad ofrece información sobre las partes del archivo que ya se han descargado. Como el usuario puede forzar al navegador a descargar el medio desde diferentes posiciones en la línea de tiempo, la información que devuelve **buffered** es un array que contiene cada parte del medio que se ha descargado, no solo la que comienza desde el inicio del medio. A los elementos del array se puede acceder mediante los métodos **end()** y **start()**. Por ejemplo, el código **buffered.start(0)** devuelve el tiempo en el que comienza la primera parte del medio y **buffered.end(0)** devuelve el tiempo en el que esa misma porción termina.

Los elementos también incluyen los siguientes métodos para manipular el medio.

play()—Este método reproduce el medio.

pause()—Este método pausa el medio.

load()—Este método carga el archivo del medio. Es útil cuando necesitamos cargar el medio de antemano en aplicaciones dinámicas.

canPlayType(tipo**)**—Este método devuelve un valor que determina si un formato de archivo es compatible con el navegador o no. El atributo **tipo** es un tipo MIME que representa el formato del medio, como video/mp4 o video/ogg. El método puede devolver tres valores dependiendo de lo seguro que esté de que puede reproducir el medio: una cadena de caracteres vacía (el formado no es compatible), el texto "maybe" (a lo mejor) y el texto "probably" (probablemente).

Esta API también incluye varios elementos para informar sobre la situación actual del medio, como el progreso al descargar el archivo, si el vídeo ha llegado al final, o si se ha pausado o se está reproduciendo, entre otros. Los siguientes son los más usados.

progress—Este evento se desencadena periódicamente para ofrecer una actualización del progreso de la descarga del medio. A la información se puede acceder a través del atributo **`buffered`**.

canplaythrough—Este evento se desencadena cuando el medio completo se puede reproducir sin interrupción. El estado se establece considerando la velocidad de descarga actual y asumiendo que seguirá siendo la misma durante el resto del proceso. Existe otro evento para este propósito llamado **`canplay`**, pero no considera toda la situación y se desencadena cuando apenas hay disponibles unos pocos cuadros.

ended—Este evento se desencadena cuando el medio termina de reproducirse.

pause—Este evento se desencadena cuando se pausa el medio.

play—Este evento se desencadena cuando el medio comienza a reproducirse.

error—Este evento se desencadena cuando ocurre un error. Se despacha al elemento **`<source>`** correspondiente con la fuente del medio que ha producido el error.

Reproductor de vídeo

Todo reproductor de vídeo necesita un panel de control con algunas herramientas básicas. La API no facilita una manera de crear estos botones o barras, tenemos que definirlas nosotros usando HTML y CSS. El siguiente ejemplo crea una interfaz con dos botones para reproducir y silenciar el vídeo, un elemento **`<div>`** para representar una barra de progreso y un deslizador para controlar el volumen.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Reproductor de Video</title>
  <link rel="stylesheet" href="reproductor.css">
```

```
  <script src="reproductor.js"></script>
</head>
<body>
  <section id="reproductor">
    <video id="medio" width="720" height="400">
      <source src="trailer.mp4">
      <source src="trailer.ogg">
    </video>
    <nav>
      <div id="botones">
        <input type="button" id="reproducir" value=">">
        <input type="button" id="silenciar" value="Silencio">
      </div>
      <div id="barra">
        <div id="progreso"></div>
      </div>
      <div id="control">
        <input type="range" id="volumen" min="0" max="1" step="0.1"
value="0.6">
      </div>
      <div class="recuperar"></div>
    </nav>
  </section>
</body>
</html>
```

Listado 8-8: *Creando un reproductor de vídeo con HTML*

Este documento también incluye recursos de dos archivos externos. Uno de estos archivos es reproductor.css con los estilos necesarios para el reproductor.

```
#reproductor {
  width: 720px;
  margin: 20px auto;
  padding: 10px 5px 5px 5px;
  background: #999999;
  border: 1px solid #666666;
  border-radius: 10px;
}
#reproducir, #silenciar {
  padding: 5px 10px;
  width: 70px;
  border: 1px solid #000000;
  background: #DDDDDD;
  font-weight: bold;
  border-radius: 10px;
}
nav {
  margin: 5px 0px;
}
#botones {
  float: left;
  width: 145px;
  height: 20px;
  padding-left: 5px;
}
```

```
#barra {
  float: left;
  width: 400px;
  height: 16px;
  padding: 2px;
  margin: 2px 5px;
  border: 1px solid #CCCCCC;
  background: #EEEEEE;
}
#progreso {
  width: 0px;
  height: 16px;
  background: rgba(0,0,150,.2);
}
.recuperar {
  clear: both;
}
```

Listado 8-9: *Diseñando el reproductor*

El código CSS del Listado 8-9 usa técnicas del modelo de caja tradicional para dibujar una caja en el centro de la pantalla que contiene todos los componentes del reproductor. Estas reglas no introducen ninguna novedad, excepto por el tamaño asignado al elemento **progreso**. Como hemos hecho en el ejemplo del Listado 6-163 (Capítulo 6), definimos el ancho inicial de este elemento como 0 píxeles para poder usarlo como una barra de progreso. El resultado se muestra en la Figura 8-2.

Figura 8-2: *Reproductor de vídeo personalizado*
© Derechos Reservados 2008, Blender Foundation/www.bigbuckbunny.org

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 8-8. Cree dos archivos para los estilos CSS y los códigos JavaScript llamados reproductor.css y reproductor.js, respectivamente. Copie el código del Listado 8-9 dentro del archivo CSS y luego copie todos los códigos JavaScript introducidos a continuación dentro del archivo JavaScript.

Como siempre, deberíamos comenzar el código JavaScript declarando las variables globales que requiere la aplicación y la función que se ejecutará tan pronto como se cargue el documento.

```
var maximo, mmedio, reproducir, barra, progreso, silenciar, volumen, 
bucle;
function iniciar() {
  maximo = 400;
  mmedio = document.getElementById("medio");
  reproducir = document.getElementById("reproducir");
  barra = document.getElementById("barra");
  progreso = document.getElementById("progreso");

  silenciar = document.getElementById("silenciar");
  volumen = document.getElementById("volumen");

  reproducir.addEventListener("click", presionar);
  silenciar.addEventListener("click", sonido);
  barra.addEventListener("click", mover);
  volumen.addEventListener("change", nivel);
}
```

Listado 8-10: *Inicializando la aplicación*

En la función **iniciar()** del Listado 8-10, creamos una referencia a cada elemento y también inicializamos la variable **maximo** para establecer el tamaño máximo de la barra de progreso (400 píxeles). Además, declaramos listeners para múltiples eventos que nos permiten responder a las acciones del usuario. Hay varias acciones a las que tenemos que prestar atención: cuando el usuario hace clic en los botones ">" (reproducir) y Silencio, cuando cambia el volumen desde el elemento **volumen** o cuando hace clic en la barra de progreso para retroceder o adelantar el vídeo. Con estos objetivos, agregamos listeners para el evento **click** a los elementos **reproducir**, **silenciar**, y **barra**, y uno para el evento **change** al elemento **volumen** para controlar el volumen. Cada vez que el usuario hace clic en uno de estos elementos o mueve el deslizador, se ejecutan las funciones correspondientes: **presionar()** para el botón ">" (reproducir), **sonido()** para el botón Silencio, **mover()** para la barra de progreso y **nivel()** para el deslizador.

La primera función que tenemos que implementar es **presionar()**. Esta función se encarga de reproducir o pausar el vídeo cuando se pulsan los botones ">" (reproducir) o "||" (pausar).

```
function presionar() {
  if (!medio.paused && !medio.ended) {
    medio.pause();
    reproducir.value = ">";
    clearInterval(bucle);
  } else {
    medio.play();
    reproducir.value = "||";
    bucle = setInterval(estado, 1000);
  }
}
```

Listado 8-11: *Reproduciendo y pausando el vídeo*

La función **presionar()** se ejecuta cuando el usuario hace clic en el botón ">" (reproducir). Este botón tiene dos propósitos: muestra el carácter ">" para reproducir el vídeo o "||" para pausarlo, de acuerdo al estado actual. Cuando el vídeo se pausa o no ha

comenzado a reproducirse, pulsar este botón reproducirá el vídeo, pero el vídeo se pausará si ya se está reproduciendo. Para determinar esta condición, el código detecta el estado del medio leyendo las propiedades **paused** y **ended**. Esto lo realiza la instrucción **if** en la primera línea de la función. Si los valores de las propiedades **paused** y **ended** son **false**, significa que el vídeo se está reproduciendo y, por lo tanto, se ejecuta el método **pause()** para pausarlo y el título del botón cambia a ">".

En esta oportunidad, aplicamos el operador **!** (NO lógico) a cada propiedad para lograr nuestro propósito. Si las propiedades devuelven **false**, el operador cambia el valor a **true**. La instrucción **if** se debe leer como "si el medio *no* se ha pausado y el medio *no* ha finalizado, entonces hacer esto ".

Si el vídeo se ha pausado o se ha terminado de reproducir, la condición es **false** y el método **play()** se ejecuta para comenzar a reproducir o continuar reproduciendo el vídeo. En este caso, también realizamos una tarea importante que es la de iniciar la ejecución de la función **estado()** cada segundo con el método **setInterval()** del objeto **Window** (ver Capítulo 6) para actualizar la barra de progreso. La siguiente es la implementación de esta función.

```
function estado() {
  if (!medio.ended) {
    var largo = parseInt(medio.currentTime * maximo / medio.duration);
    progreso.style.width = largo + "px";
  } else {
    progreso.style.width = "0px";
    reproducir.value = ">";
    clearInterval(bucle);
  }
}
```

Listado 8-12: *Actualizando la barra de progreso*

La función **estado()** se ejecuta cada segundo mientras se reproduce el vídeo. En esta función también tenemos una instrucción **if** para controlar el estado del vídeo. Si la propiedad **ended** devuelve **false**, calculamos la longitud que debería tener la barra de progreso en píxeles y declaramos el nuevo tamaño para el elemento **<div>** que la representa. Pero si el valor de la propiedad **ended** es **true** (lo cual significa que el vídeo ha finalizado), declaramos el tamaño de la barra de progreso nuevamente a **0** píxeles, cambiamos el texto del botón a ">" (reproducir) y cancelamos el bucle con el método **clearInterval()**. Después de esto, la función **estado()** ya no se ejecuta.

Para calcular el valor actual, necesitamos el valor de la propiedad **currentTime** para saber qué parte del vídeo se está reproduciendo, el valor de la propiedad **duration** para saber la duración del vídeo y el valor de la variable **maximo** para obtener el tamaño máximo permitido para la barra de progreso. La fórmula es una regla de tres simple. Tenemos que multiplicar el tiempo actual por el tamaño máximo y dividir el resultado por la duración (**tiempo-actual × maximo / duración**). El resultado es el nuevo tamaño en píxeles del elemento **<div>** que representa la barra de progreso.

La función que responde al evento **click** del elemento **reproducir** (el botón de reproducir) ya se ha creado, por lo que es hora de hacer lo mismo para el evento **click** de la barra de progreso. A este método lo llamamos **mover()**.

```
function mover(evento) {
  if (!medio.paused && !medio.ended) {
    var ratonX = evento.offsetX - 2;
    if (ratonX < 0) {
      ratonX = 0;
    } else if (ratonX > maximo) {
      ratonX = maximo;
    }
    var tiempo = ratonX * medio.duration / maximo;
    medio.currentTime = tiempo;
    progreso.style.width = ratonX + "px";
  }
}
```

Listado 8-13: *Reproduciendo el vídeo desde la posición seleccionada por el usuario*

En la función **`iniciar()`**, habíamos agregado un listener al elemento **`barra`** para el evento **`click`** con la intención de responder cada vez que el usuario quiere comenzar a reproducir el vídeo desde una nueva posición. Cuando este evento se desencadena, se ejecuta la función **`mover()`**. Esta función comienza con una instrucción **`if`**, como las anteriores funciones, pero esta vez el objetivo es llevar a cabo la acción solo cuando el vídeo se está reproduciendo. Si las propiedades **`paused`** y **`ended`** devuelven **`false`**, significa que el vídeo se está reproduciendo y que podemos ejecutar el código.

Para calcular el tiempo en el que se debe seguir reproduciendo el vídeo, obtenemos la posición del ratón desde la propiedad **`offsetX`** (ver Listado 6-163, Capítulo 6), calculamos el tamaño de la barra de progreso considerando el relleno del elemento **`barra`** y luego convertimos este tamaño en segundos para comenzar a reproducir el vídeo desde la nueva ubicación. El valor en segundos que representa la posición del ratón en la línea de tiempo se calcula con la fórmula **`ratonX × medio.duration / maximo`**. Una vez que obtenemos este valor, tenemos que asignarlo a la propiedad **`currentTime`** para comenzar a reproducir el vídeo en esa posición. Al final, la posición del ratón se asigna a la propiedad **`width`** del elemento **`progreso`** para reflejar el nuevo estado del vídeo en la pantalla.

Además de estas funciones, necesitamos dos más para controlar el audio del medio. La primera es la función **`sonido()`**, asignada como listener del evento **`click`** del botón Silencio.

```
function sonido() {
  if (silenciar.value == "Silencio") {
    medio.muted = true;
    silenciar.value = "Sonido";
  } else {
    medio.muted = false;
    silenciar.value = "Silencio";
  }
}
```

Listado 8-14: *Activando y desactivando el sonido con la propiedad* `muted`

La función del Listado 8-14 activa o desactiva el sonido del medio dependiendo del valor del atributo **`value`** del botón Silencio. El botón muestra diferentes textos de acuerdo a la

situación. Si el valor actual es "Silencio", el sonido se desactiva y el título del botón cambia a "Sonido". En caso contrario, el sonido se activa y el título del botón cambia a "Silencio".

Cuando el sonido está activo, el volumen se puede controlar a través del elemento **volumen**, localizado al final de la barra de progreso. El elemento desencadena el evento **change** cada vez que se modifica su valor (cada vez que se desplaza el control). A la función que responde al evento la llamamos **nivel()**.

```
function nivel() {
  medio.volume = volumen.value;
}
```

Listado 8-15: *Controlando el volumen*

La función del Listado 8-15 asigna el valor del atributo **value** del elemento **<input>** que representa el control a la propiedad **volume** del medio. Lo único que tenemos que recordar es que esta propiedad acepta valores entre **0.0** y **1.0**. Los números fuera de esta rango devolverán un error.

Con esta pequeña función el código del reproductor está casi listo, solo nos queda responder al evento **load** para iniciar la aplicación cuando se descarga el documento.

```
window.addEventListener("load", iniciar);
```

Listado 8-16: *Respondiendo al evento* `load` *para iniciar la aplicación*

Hágalo usted mismo: copie todo el código JavaScript introducido desde el Listado 8-10 dentro del archivo reproductor.js. Abra el documento del Listado 8-8 en su navegador y haga clic en el botón ">" (reproducir). Intente ejecutar la aplicación en diferentes navegadores.

8.4 Subtítulos

Los subtítulos son texto que se muestra sobre el reproductor mientras se reproduce el vídeo. Este sistema se ha utilizado durante décadas en televisión y en diferentes medios de distribución de vídeo, pero hasta este momento no había sido fácil incluirlo en la Web. HTML ofrece el siguiente elemento para simplificar su implementación.

<track>—Este elemento agrega subtítulos a un vídeo.

El elemento tiene que incluirse dentro de un elemento **<video>** o **<audio>**. Para especificar la fuente, el tipo y el modo en el que se mostrarán los subtítulos en la pantalla, el elemento **<track>** ofrece los siguientes atributos.

src—Este atributo declara la URL del archivo que contiene el texto de los subtítulos. El formato de este archivo puede ser cualquiera de los que admiten los navegadores, pero la especificación declara al formato WebVTT como el oficial para este elemento.

srclang—Este atributo declara el idioma del texto. Trabaja con los mismos valores que el atributo **lang** del elemento **<html>** estudiado en el Capítulo 2.

default—Este atributo declara la pista (*track*) que queremos incluir por defecto. Si solo se facilita un elemento **<track>**, este atributo se puede usar para activar los subtítulos.

label—Este atributo facilita un título para la pista. Si se incluyen varios elementos **<track>**, este atributo ayuda a los usuarios a encontrar el correcto.

kind—Este atributo declara el tipo de contenido asignado a una pista. Los valores disponibles son **subtitles** (para subtítulos), **captions** (para subtítulos que representan sonido), **descriptions** (destinado a sintetizado de audio), **chapters** (para navegación entre capítulos) y **metadata** (para información adicional que no se muestra en la pantalla). El valor por defecto es **subtitles** (subtítulos).

El elemento **<track>** trabaja con los elementos **<video>** y **<audio>** para agregar subtítulos o mostrar información adicional para nuestros vídeos y pistas de audio.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Subtítulos</title>
</head>
<body>
  <section>
    <video width="720" height="400" controls>
      <source src="trailer.mp4">
      <source src="trailer.ogg">
      <track src="subtitulos.vtt" srclang="es" default>
    </video>
  </section>
</body>
</html>
```

Listado 8-17: *Agregando subtítulos con el elemento* <track>

IMPORTANTE: el elemento **<track>** no permite la construcción de aplicaciones de origen cruzado, y, por lo tanto, deberá subir todos los archivos a su propio servidor y ejecutar la aplicación bajo el mismo dominio (incluidos el documento HTML, los vídeos y los subtítulos). Otra alternativa es usar un servidor local, como el que se instala mediante la aplicación MAMP (ver Capítulo 1 para más información). Esta situación se puede evitar con la tecnología CORS y el atributo **crossorigin**, como explicaremos en el Capítulo 11.

En el ejemplo del Listado 8-17 declaramos un solo elemento **<track>**. El idioma de la fuente se ha declarado como Español, y se ha incluido el atributo **default** para declarar esta pista como la pista por defecto y de este modo activar los subtítulos. La fuente de este elemento se ha declarado como un archivo en formato WebVTT (subtitulos.vtt). WebVTT son las siglas del nombre Web Video Text Tracks, un formato estándar para subtítulos. Los archivos de este formato son simplemente texto con una estructura especial, tal como ilustra el siguiente ejemplo.

```
WEBVTT

00:02.000 --> 00:07.000
Bienvenido
al elemento &lt;track&gt;!

00:10.000 --> 00:15.000
Este es un ejemplo simple.

00:17.000 --> 00:22.000
Varias pistas pueden ser usadas simultaneamente

00:22.000 --> 00:25.000
para ofrecer textos en diferentes lenguajes.

00:27.000 --> 00:30.000
Hasta luego!
```

Listado 8-18: *Definiendo un archivo WebVTT*

El Listado 8-18 muestra la estructura de un archivo WebVTT. La primera línea con el texto "WEBVTT" es obligatoria, así como las líneas vacías entre cada declaración. Las declaraciones se llaman *cues* (señales), y requieren la sintaxis minutos:segundos.milisegundos para indicar el tiempo de inicio y finallización en el que aparecerán. Este es un formato rígido; siempre tenemos que respetar la estructura que se muestra en el ejemplo del Listado 8-18 y declarar cada parámetro con la misma cantidad de dígitos (dos para minutos, dos para segundos y tres para milisegundos).

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 8-17. Cree un archivo de texto con las declaraciones del Listado 8-18 y el nombre subtitulos.vtt. Suba estos archivos y el vídeo a su servidor, o copie los archivos en su servidor local (ver MAMP en el Capítulo 1), y abra el documento HTML en su navegador. Debería ver el vídeo reproduciéndose con subtítulos y un botón a un lado para activarlos o desactivarlos.

Las *cues* (las líneas de texto en un archivo de subtítulos) pueden incluir las etiquetas especiales **<b>**, **<i>**, **<u>**, **<v>** y **<c>**. Las tres primeras son para otorgar énfasis, como en HTML, mientras que la etiqueta **<v>** declara a quién pertenece el texto (quien habla) y la etiqueta **<c>** nos permite asignar estilos usando CSS.

```
WEBVTT

00:02.000 --> 00:07.000
<i>Bienvenido</i>
al elemento &lt;track&gt;!

00:10.000 --> 00:15.000
<v Roberto>Este es un ejemplo <c.titulos>simple</c>.

00:17.000 --> 00:22.000
<v Martin>Se pueden usar varias pistas simultaneamente
```

```
00:22.000 --> 00:25.000
<v Martin>para ofrecer textos en diferentes lenguajes.

00:27.000 --> 00:30.000
<b>Hasta luego!</b>
```

Listado 8-19: *Incluyendo etiquetas en un archivo WebVTT*

El formato WebVTT usa un seudoelemento para referenciar las *cues*. Este seudoelemento, llamado **`::cue`**, puede recibir un selector de una clase entre paréntesis. En el ejemplo del Listado 8-19, la clase se ha llamado *titulos* (<c.titulos>), por lo que el selector CSS se debe escribir como **`::cue(.titulos)`**, como en el siguiente ejemplo.

```
::cue(.titulos){
  color: #990000;
}
```

Listado 8-20: *Declarando estilos para un archivo WebVTT*

IMPORTANTE: solo un reducido grupo de propiedades CSS, como **`color`**, **`background`**, y **`font`**, están disponibles para este seudoelemento; el resto se ignoran.

El formato WebVTT también ofrece la posibilidad de alinear y posicionar cada *cue* usando los siguientes parámetros y valores.

align—Este parámetro alinea la *cue* en relación al centro del espacio que cubre el medio. Los valores disponibles son **`start`**, **`middle`**, y **`end`**.

vertical—Este parámetro cambia la orientación a vertical y ordena la *cue* de acuerdo a dos valores: **`rl`** (derecha a izquierda) o **`lr`** (izquierda a derecha).

position—Este parámetro declara la posición de la *cue* en columnas. El valor se puede expresar como un porcentaje o como un número de **`0`** a **`9`**. La posición se declara de acuerdo a la orientación.

line—Este parámetro declara la posición de la *cue* en filas. El valor se puede expresar en porcentaje o como un número de **`0`** a **`9`**. En una orientación horizontal, la posición que se declara es vertical, y viceversa. Los números positivos declaran la posición desde un lado y los números negativos desde el otro, dependiendo de la orientación.

size—Este valor declara el tamaño de la *cue*. El valor se puede declarar en porcentaje y se determina a partir del ancho del medio.

Estos parámetros y sus correspondientes valores se declaran al final de la *cue* separados por dos puntos. Se pueden hacer varias declaraciones para la misma *cue*, como muestra el siguiente ejemplo.

```
WEBVTT

00:02.000 --> 00:07.000 align:start position:5%
```

```
<i>Bienvenido</i>
al elemento &lt;track&gt;!
```

Listado 8-21: *Configurando cues*

Hágalo usted mismo: usando el archivo WebVTT creado en el Listado 8-18 intente combinar diferentes parámetros en las mismas *cues* para ver los efectos disponibles para este formato.

8.5 API TextTrack

La API TextTrack se definió para ofrecer acceso desde JavaScript al contenido de las pistas usadas como subtítulos. La API incluye un objeto llamado **`TextTrack`** para devolver esta información. Existen dos maneras de obtener este objeto: desde el elemento de medios o desde el elemento **`<track>`**.

textTracks—Esta propiedad contiene un array con los objetos **`TextTrack`** de cada pista del medio. Los objetos **`TextTrack`** se almacenan en el array en orden secuencial.

track—Esta propiedad devuelve el objeto **`TextTrack`** de la pista especificada.

Si el medio (vídeo o audio) contiene varios elementos **`<track>`**, puede resultar más fácil encontrar la pista que buscamos accediendo al array **`textTracks`**.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Trabajando con Pistas</title>
  <script>
    function iniciar() {
      var video = document.getElementById("medio");
      var pista1 = video.textTracks[0];
      var pista = document.getElementById("pista");
      var pista2 = pista.track;
      console.log(pista1);
      console.log(pista2);
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <video id="medio" width="720" height="400" controls>
      <source src="trailer.mp4">
      <source src="trailer.ogg">
      <track id="pista" label="Subtítulos en Español"
src="subtitulos.vtt" srclang="es" default>
    </video>
  </section>
</body>
</html>
```

Listado 8-22: *Obteniendo el objeto* `TextTrack`

El documento del Listado 8-22 demuestra cómo acceder al objeto **`TextTrack`** usando ambas propiedades. La función **`iniciar()`** crea una referencia al elemento **`<video>`** para obtener el objeto **`TextTrack`** accediendo al array **`textTracks`** con el índice **`0`** y luego se obtiene nuevamente el mismo objeto desde el elemento **`<track>`** usando la propiedad **`track`**. Finalmente, el contenido de ambas variables se imprime en la consola.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 8-22. Como en ejemplos anteriores, esta aplicación no trabaja en un ordenador local; debe subir los archivos a su servidor o moverlos a un servidor local para probarlos.

Leyendo pistas

Una vez que obtenemos el objeto **`TextTrack`** de la pista con la que queremos trabajar, podemos acceder a sus propiedades.

kind—Esta propiedad devuelve el tipo de pista, tal como se ha especificado mediante el atributo **`kind`** del elemento **`<track>`** (**`subtitles`**, **`captions`**, **`descriptions`**, **`chapters`** o **`metadata`**).

label—Esta propiedad devuelve la etiqueta de la pista, tal como se ha especificado con el atributo **`label`** del elemento **`<track>`**.

language—Esta propiedad devuelve el idioma de la pista, tal como se ha especificado mediante el atributo **`srclang`** del elemento **`<track>`**.

mode—Esta propiedad devuelve o declara el modo de la pista. Los valores disponibles son **`disabled`** (desactivada), **`hidden`** (oculta) y **`showing`** (mostrar). Se puede usar para intercambiar pistas.

cues—Esta propiedad es un array que contiene todas las *cues* de la pista.

activeCues—Esta propiedad devuelve las *cues* que actualmente se están mostrando en pantalla (la anterior, la actual y la siguiente).

A partir de las propiedades del objeto **`TextTrack`** podemos acceder a toda la información almacenada en la pista.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Trabajando con Pistas</title>
  <style>
    #reproductor, #info {
      float: left;
    }
  </style>
  <script>
    function iniciar() {
      var info = document.getElementById("info");
      var pista = document.getElementById("pista");
      var obj = pista.track;
```

```
      var lista = "";
      lista += "<br>Tipo: " + obj.kind;
      lista += "<br>Etiqueta: " + obj.label;
      lista += "<br>Idioma: " + obj.language;
      info.innerHTML = lista;
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section id="reproductor">
    <video id="medio" width="720" height="400" controls>
      <source src="trailer.mp4">
      <source src="trailer.ogg">
      <track id="pista" label="Subtítulos en Español"
src="subtitulos.vtt" srclang="es" default>
    </video>
  </section>
  <aside id="info"></aside>
</body>
</html>
```

Listado 8-23: *Mostrando la información de la pista en pantalla*

El documento del Listado 8-23 incluye estilos para las dos columnas creadas por los elementos **<section>** y **<aside>**, y el código JavaScript para obtener y mostrar los datos del elemento **<track>**. Esta vez, obtenemos el objeto **TextTrack** desde la propiedad **track** y lo almacenamos en la variable **obj**. Desde esta variable, leemos los valores de las propiedades del objeto y generamos el texto para mostrarlos en pantalla.

Leyendo *cues*

Además de las propiedades aplicadas en el último ejemplo, también contamos con una propiedad importante llamada **cues**. Esta propiedad contiene un array con objetos **TextTrackCue** que representan cada *cue* de la pista.

```
<script>
  function iniciar() {
    var info = document.getElementById("info");
    var pista = document.getElementById("pista");
    var obj = pista.track;
    var lineas = obj.cues;

    var lista = "";
    for (var f = 0; f < lineas.length; f++) {
      lista += lineas[f].text + "<br>";
    }
    info.innerHTML = lista;
  }
  window.addEventListener("load", iniciar);
</script>
```

Listado 8-24: *Mostrando cues en la pantalla*

La nueva función **iniciar()** del Listado 8-24 accede a cada *cue* usando un bucle **for**. Dentro del bucle, los valores del array se agregan a la variable **lista** junto con un elemento **
** para mostrar las *cues* una por línea, y luego todo el texto se inserta dentro del elemento **<aside>** para mostrarse en pantalla.

Los objetos **TextTrackCue** incluyen propiedades con la información de cada *cue*. En nuestro ejemplo, mostramos el contenido de la propiedad **text**. La siguiente es una lista de las propiedades disponibles.

text—Esta propiedad devuelve el texto de la *cue*.

startTime—Esta propiedad devuelve el tiempo de inicio de la *cue* en segundos.

endTime—Esta propiedad devuelve el tiempo de finalización de la *cue* en segundos.

vertical—Esta propiedad devuelve el valor del parámetro **vertical**. Si el parámetro no se ha definido, el valor que devuelve es una cadena de caracteres vacía.

line—Esta propiedad devuelve el valor del parámetro **line**. Si el parámetro no se ha definido, el valor se devuelve por defecto.

position—Esta propiedad devuelve el valor del parámetro **position**. Si el parámetro no se ha especificado, el valor se devuelve por defecto.

size—Esta propiedad devuelve el valor del parámetro **size**. Si el parámetro no se ha definido, el valor se devuelve por defecto.

align—Esta propiedad devuelve el valor del parámetro **align**. Si el parámetro no se ha definido, el valor se devuelve por defecto.

El siguiente código actualiza el ejemplo anterior para agregar los tiempos de inicio de cada *cue*.

```
<script>
  function iniciar() {
    var info = document.getElementById("info");
    var pista = document.getElementById("pista");
    var obj = pista.track;
    var lineas = obj.cues;
    var lista = "";
    for (var f = 0; f < lineas.length; f++) {
      var linea = lineas[f];
      lista += linea.startTime + " - ";
      lista += linea.text + "<br>";
    }
    info.innerHTML = lista;
  }
  window.addEventListener("load", iniciar);
</script>
```

Listado 8-25: *Mostrando información acerca de las cues*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 8-23. Suba los archivos a su servidor o muévalos a su servidor local y abra el documento en su navegador. Para trabajar con *cues*, reemplace la función **iniciar()** en el documento por la que quiere probar. Las *cues* y sus valores se deberían mostrar en el lado derecho del vídeo.

Agregando pistas

El objeto **`TextTrack`** que representa una pista no solo tiene propiedades, sino también métodos con los que podemos crear nuevas pistas y *cues* desde JavaScript.

addTextTrack(tipo, etiqueta, idioma**)**—Este método crea una nueva pista para el medio y devuelve el objeto **`TextTrack`** correspondiente. Los atributos son los valores de los atributos para la pista (solo **tipo** es obligatorio).

addCue(objeto**)**—Este método agrega una nueva *cue* a la pista. El atributo **objeto** es un objeto **`TextTrackCue`** que devuelve el constructor **`VTTCue()`**.

removeCue(objeto**)**—Este método elimina una *cue* de la pista. El atributo **objeto** es un objeto **`TextTrackCue`** que devuelve el objeto **`TextTrack`**.

Para agregar *cues* a la pista, tenemos que proveer un objeto **`TextTrackCue`**. La API incluye un constructor para crear este objeto.

VTTCue(inicio, finalización, texto**)**—Este constructor devuelve un objeto **`TextTrackCue`** para usar con el método **`addCue()`**. Los atributos representan los datos para la *cue*.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Trabajando con Pistas</title>
  <script>
    function iniciar(){
      var linea;
      var lineas = [
        { start: 2.000, end: 7.000, text: "Bienvenido"},
        { start: 10.000, end: 15.000, text: "Este es un ejemplo"},
        { start: 15.001, end: 20.000, text: "de como agregar"},
        { start: 20.001, end: 25.000, text: "una nueva pista."},
        { start: 27.000, end: 30.000, text: "Hasta Luego!"},
      ];
      var video = document.getElementById("medio");

      var nuevapista = video.addTextTrack("subtitles");
      nuevapista.mode = "showing";
      for (var f = 0; f < lineas.length; f++) {
        linea = new VTTCue(lineas[f].start, lineas[f].end, lineas[f].text);
        nuevapista.addCue(linea);
      }
      video.play();
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <video id="medio" width="720" height="400" controls>
      <source src="trailer.mp4">
```

```
      <source src="trailer.ogg">
    </video>
  </section>
</body>
</html>
```

Listado 8-26: *Agregando pistas y cues desde JavaScript*

En el Listado 8-26, comenzamos la función **iniciar()** definiendo el array **cues** con cinco *cues* para nuestra pista. Las *cues* se declaran como elementos del array. Cada *cue* es un objeto con las propiedades **start**, **end** y **text**. Los valores de los tiempos de inicio y finalización no usan la sintaxis del archivo WebVTT; en cambio, se tienen que declarar en segundos con números decimales.

Las *cues* se pueden agregar a una pista existente o a una nueva. En nuestro ejemplo, hemos creado una nueva pista de tipo **subtitles** usando el método **addTextTrack()**. También tenemos que declarar el modo (**mode**) de esta pista como **showing**, para pedirle al navegador que muestre la pista en la pantalla. Cuando la pista está lista, todas las *cues* del array **cues** se convierten en objetos **TextTrackCue** y se agregan a la pista con el método **addCue()**.

Hágalo usted mismo: actualice su documento con el código del Listado 8-26 y ábralo en su navegador. Debería ver la nueva pista sobre el vídeo. Recuerde subir los archivos a su servidor o moverlos a su servidor local.

Capítulo 9
API Stream

9.1 Capturando medios

La API Stream nos permite acceder a las transmisiones de medios en el dispositivo. Las transmisiones más comunes son las que se realizan con la cámara y el micrófono, pero esta API se ha definido para facilitar acceso a cualquier otra fuente que produce transmisiones de vídeo o audio.

La API define el objeto **`MediaStream`** para referenciar las transmisiones de medios y el objeto **`LocalMediaStream`** para transmisiones generadas por dispositivos locales. Estos objetos tienen una entrada representada por el dispositivo y una salida representada por elementos como **<video>** y **<audio>**, además de otras API. Para obtener el objeto **`LocalMediaStream`** que representa una transmisión, la API incluye un objeto llamado **`MediaDevices`** que, entre otros, incluye el siguiente método.

getUserMedia(restricciones**)**—Este método solicita el permiso del usuario para acceder a la transmisión del vídeo o audio y en respuesta genera un objeto **`LocalMediaStream`** que representa la transmisión. El atributo **restricciones** es un objeto con dos propiedades **`video`** y **`audio`** que indican el tipo de medio a capturar.

El método **`getUserMedia()`** realiza una operación asíncrona, lo que significa que intentará acceder a la transmisión mientras se sigue ejecutando el resto del código. Para este propósito, el método devuelve un objeto **`Promise`** con el que informar del resultado. Este es un objeto específicamente diseñado para controlar operaciones asíncronas. El objeto incluye dos métodos con los que procesar la respuesta.

then(función**)**—Este método se ejecuta mediante una operación asíncrona en caso de éxito. Si la operación se realiza correctamente, se llama a este método y se ejecuta la función especificada por el atributo.

catch(función**)**—Este método se ejecuta mediante una operación asíncrona en caso de error. Si se produce un error en la operación, se llama a este método y se ejecuta la función especificada por el atributo.

Cuando se accede a una transmisión, el método **`then()`** envía a la función la referencia del objeto **`LocalMediaStream`** que representa la transmisión. Para mostrarla al usuario, tenemos que asignar este objeto a un elemento **<video>** o **<audio>**. Para este propósito, los objetos **`Element`** que representan estos elementos incluyen la siguiente propiedad.

srcObject—Esta propiedad declara o devuelve el objeto **`MediaStream`** que representa la transmisión.

El dispositivo más común al que podemos acceder para transmitir vídeo es la cámara. El siguiente ejemplo ilustra cómo acceder a la cámara y mostrar la transmisión en la pantalla.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Stream</title>
  <script>
    function iniciar() {
      var promesa = navigator.mediaDevices.getUserMedia({video: true});
      promesa.then(exito);
      promesa.catch(mostrarerror);

      function exito(transmision) {
        var video = document.getElementById("medio");
        video.srcObject = transmision;
        video.play();
      }
      function mostrarerror(error) {
        console.log("Error: " + error.name);
      }
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <video id="medio"></video>
  </section>
</body>
</html>
```

Listado 9-1: *Accediendo a la cámara*

La función **iniciar()** del Listado 9-1 obtiene la transmisión desde la cámara usando el método **getUserMedia()**. Esto genera una solicitud para el usuario. Si el usuario permite a nuestra aplicación acceder a la cámara, se llama al método **then()** del objeto **Promise** y se ejecuta la función **exito()**. Esta función recibe el objeto **LocalMediaStream** generado por el método **getUserMedia()** y lo asigna al elemento **<video>** en el documento mediante la propiedad **srcObject**.

En caso de error, se llama al método **catch()** del objeto **Promise** y se ejecuta la función **mostrarerror()**. La función recibe un objeto con información acerca del error. El error más común es PermissionDeniedError, que se genera cuando el usuario deniega acceso al medio o el medio no está disponible por otras razones.

En este ejemplo no hemos tenido que declarar al atributo **src** del elemento **<video>** debido a que la fuente será la transmisión de vídeo capturada por el código JavaScript, pero podríamos haber declarado los atributos **width** y **height** para cambiar el tamaño del vídeo en la pantalla.

IMPORTANTE: el método **getUserMedia()** solo se puede ejecutar desde un servidor y el servidor debe ser seguro, lo que significa que la aplicación no funcionará a menos que subamos los archivos a un servidor que utiliza el protocolo https. Si no posee un servidor seguro, puede probar los ejemplos en un servidor local instalado con una aplicación como MAMP (ver Capítulo 1).

Lo básico: el objeto **`MediaDevices`** que facilita el método **`getUserMedia()`** pertenece al objeto **`Navigator`** (vea el Capítulo 6). Cuando el navegador crea el objeto **`Window`**, almacena una referencia del objeto **`Navigator`** en una propiedad llamada **`navigator`** y el objeto **`MediaDevices`** en una propiedad llamada **`mediaDevices`**. Esta es la razón por la que en el ejemplo del Listado 9-1 usamos estas propiedades para llamar al método **`getUserMedia()`** (**`navigator.mediaDevices.getUserMedia()`**).

El objeto MediaStreamTrack

Los objetos **`MediaStream`** (y, por lo tanto, los objetos **`LocalMediaStream`**) contienen objetos **`MediaStreamTrack`** que representan cada pista de medios (normalmente una para vídeo y otra para audio). Los objetos **`MediaStream`** incluyen los siguientes métodos para obtener los objetos **`MediaStreamTrack`**.

getVideoTracks()—Este método devuelve un array con objetos **`MediaStreamTrack`** que representan las pistas de vídeo incluidas en la transmisión.

getAudioTracks()—Este método devuelve un array con objetos **`MediaStreamTrack`** que representan las pistas de audio incluidas en la transmisión.

Los objetos **`MediaStreamTrack`** que devuelven estos métodos incluyen propiedades y méto-dos para obtener información y controlar la pista de vídeo o audio. Los siguientes son los más usados.

enabled—Esta propiedad devuelve **`true`** o **`false`** de acuerdo al estado de la pista. Si la pista aún se encuentra asociada a la fuente, el valor es **`true`**.

kind—Esta propiedad devuelve el tipo de fuente que representa la pista. Los valores disponibles son **`audio`** y **`video`**.

label—Esta propiedad devuelve el nombre de la fuente de la pista.

stop()—Este método detiene la pista.

Si queremos recoger información acerca de la transmisión, tenemos que obtener la transmisión con el método **`getUserMedia()`** como hemos hecho en el ejemplo anterior, obtener referencias a las pistas con el método **`getVideoTracks()`** y luego leer los valores de la pista que se devuelve.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Stream</title>
  <style>
    section {
      float: left;
    }
  </style>
  <script>
    function iniciar() {
      var promesa = navigator.mediaDevices.getUserMedia({video: true});
```

```
      promesa.then(exito);
      promesa.catch(mostrarerror);

      function exito(transmision) {
        var video = document.getElementById("video");
        video.srcObject = transmision;
        video.play();
        var cajadatos = document.getElementById("cajadatos");
        var pistasvideo = transmision.getVideoTracks();
        var pista = pistasvideo[0];
        var datos = "";
        datos += "<br> Habilitado: " + pista.enabled;
        datos += "<br> Tipo: " + pista.kind;
        datos += "<br> Dispositivo: " + pista.label;
        cajadatos.innerHTML = datos;
      }
      function mostrarerror(error) {
        console.log("Error: " + error.name);
      }
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <video id="video"></video>
  </section>
  <section id="cajadatos"></section>
</body>
</html>
```

Listado 9-2: *Mostrando información de la transmisión*

El método **`getVideoTracks()`** del objeto **`MediaStream`** devuelve un array. Debido a que la cámara contiene una sola pista de vídeo, para obtener una referencia a esta pista tenemos que leer el elemento al índice `0` (la primera pista en la lista).

En el documento del Listado 9-2, usamos el mismo código del ejemplo anterior para acceder a la transmisión, pero esta vez la pista de la cámara se almacena en la variable **`pista`**, y luego cada una de sus propiedades se muestran en la pantalla.

IMPORTANTE: las pistas mencionadas aquí son pistas de vídeo o audio. Estos tipos de pistas no tienen nada que ver con las pistas de subtítulos que hemos estudiado en el Capítulo 8.

Una vez que obtenemos la pista, podemos detenerla con el método **`stop()`**. En el siguiente ejemplo, agregamos un botón a la interfaz para detener la transmisión cuando lo pulsa el usuario. El listener del evento **`click`** para el botón solo se agrega si el código es capaz de acceder a la cámara. En caso contrario, el botón no hace nada.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
```

```
  <title>API Stream</title>
  <script>
    function iniciar() {
      var promesa = navigator.mediaDevices.getUserMedia({video: true});
      promesa.then(exito);
      promesa.catch(mostrarerror);
    }
    function exito(transmision) {
      var boton = document.getElementById("boton");
      boton.addEventListener("click", function() {
detenertransmision(transmision) });
      var video = document.getElementById("video");
      video.srcObject = transmision;
      video.play();
    }
    function mostrarerror(evento) {
      console.log("Error: " + error.name);
    }
    function detenertransmision(transmision) {
      var pistasvideo = transmision.getVideoTracks();
      var pista = pistasvideo[0];
      pista.stop();
      alert("Transmisión Cancelada");
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <video id="video"></video>
  </section>
  <nav>
    <button type="boton" id="boton">Apagar</button>
  </nav>
</body>
</html>
```

Listado 9-3: *Deteniendo la transmisión*

Además de las propiedades y los métodos introducidos anteriormente, los objetos **MediaStreamTrack** también ofrecen eventos para informar del estado de la pista.

muted—Este evento se desencadena cuando la pista no puede facilitar datos.

unmuted—Este evento se desencadena tan pronto como la pista comienza a proveer datos nuevamente.

ended—Este evento se desencadena cuando la pista ya no puede proveer datos. Esto puede deberse a varias razones, desde el usuario que niega el acceso a la transmisión al uso del método **stop()**.

IMPORTANTE: estos eventos están destinados a utilizarse para controlar transmisiones remotas, un proceso que estudiaremos en el Capítulo 24.

Capítulo 10
API Fullscreen

10.1 Aplicaciones modernas

La Web se ha convertido en una plataforma multimedia y multipropósito en la que todo es posible. Con tantas nuevas aplicaciones, la palabra navegación ha perdido significado. Las herramientas de la ventana del navegador no solo ya no son necesarias en muchas circunstancias, sino que a veces se interponen entre el usuario y la aplicación. Por esta razón, los navegadores incluyen la API Fullscreen.

Pantalla Completa

La API Fullscreen nos permite expandir cualquier elemento en el documento hasta ocupar la pantalla completa. Como resultado, la interfaz del navegador se oculta y permite al usuario centrar su atención en nuestros vídeos, imágenes, aplicaciones o videojuegos.

La API provee propiedades, métodos y eventos para llevar a un elemento al modo de pantalla completa, salir de ese modo, y obtener información del elemento y el documento que están participando en el proceso.

fullscreenElement—Esta propiedad devuelve una referencia al elemento que se está mostrando en pantalla completa. Si ningún elemento está en pantalla completa, la propiedad devuelve el valor `null`.

fullscreenEnabled—Esta es una propiedad booleana que devuelve `true` cuando el documento puede activar la pantalla completa o `false` en caso contrario.

requestFullscreen()—Este método se aplica a todos los elementos del documento. El método activa el modo de pantalla completa para el elemento.

exitFullscreen()—Este método se aplica al documento. Si un elemento se encuentra en modo de pantalla completa, este método cancela el modo y devuelve el foco de nuevo a la ventana del navegador.

La API también ofrece los siguientes eventos.

fullscreenchange—El documento desencadena este evento cuando un elemento entra o sale del modo de pantalla completa.

fullscreenerror—Este evento se desencadena por el elemento en caso de error (el modo de pantalla completa no está disponible para ese elemento o para el documento).

El método `requestFullscreen()` y el evento `fullscreenerror` están asociados con los elementos del documento, pero el resto de las propiedades, métodos y eventos son parte del objeto `Document` y, por lo tanto, son accesibles desde la propiedad `document`.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Pantalla Completa</title>
  <script>
    var video;
    function iniciar() {
      video = document.getElementById("medio");
      video.addEventListener("click", expandir);
    }
    function expandir() {
      if (!document.webkitFullscreenElement) {
        video.webkitRequestFullscreen();
        video.play();
      }
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <video id="medio" width="720" height="400" poster="poster.jpg">
      <source src="trailer.mp4">
      <source src="trailer.ogg">
    </video>
  </section>
</body>
</html>
```

Listado 10-1: *Llevando al elemento* `<video>` *a pantalla completa*

La función **`iniciar()`** del Listado 10-1 agrega un listener para el evento **`click`** al elemento **`<video>`**. Como resultado, la función **`expandir()`** se ejecuta cada vez que el usuario hace clic en el vídeo. En esta función, usamos la propiedad **`fullscreenElement`** para detectar si un elemento ya se encuentra en pantalla completa o no, y si no, el vídeo se lleva a pantalla completa con el método **`requestFullscreen()`**. Al mismo tiempo, el vídeo se reproduce con el método **`play()`**.

IMPORTANTE: esta es una API experimental. Las propiedades, los métodos y los eventos se tienen que declarar con prefijos hasta que se implementa la especificación final.
En el caso de Google Chrome, en lugar de los nombres originales tenemos que declarar **`webkitRequestFullscreen()`**, **`webkitExitFullscreen()`**, **`webkitfullscreenchange`**, **`webkitfullscreenerror`** y **`webkitFullscreenElement`**.
Para Mozilla Firefox, los nombres son **`mozRequestFullScreen()`**, **`mozCancelFullScreen()`**, **`mozfullscreenchange`**, **`mozfullscreenerror`** y **`mozFullScreenElement`**.

Estilos de pantalla completa

Los navegadores ajustan las dimensiones del elemento **<video>** al tamaño de la pantalla automáticamente, pero para otros elementos las dimensiones originales se preservan y el espacio libre en pantalla se llena con un fondo negro. Por esta razón, CSS incluye una seudoclase llamada **:full-screen** para modificar los estilos de un elemento cuando se lleva a pantalla completa.

IMPORTANTE: la última especificación declara esta seudoclase con el nombre sin el guion (**:fullscreen**), pero, en el momento de escribir estas líneas, navegadores como Mozilla Firefox y Google Chrome solo han implementado la especificación anterior que trabaja con el nombre mencionado en este capítulo (**:full-screen**). También debemos recordar agregar el correspondiente prefijo (**:-webkit-full-screen**, **:-moz-full-screen**, etc.).

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Pantalla Completa</title>
  <style>
    #reproductor:-webkit-full-screen, #reproductor:-webkit-full-screen
#medio {
      width: 100%;
      height: 100%;
    }
  </style>
  <script>
    var video, reproductor;
    function iniciar() {
      video = document.getElementById("medio");
      reproductor = document.getElementById("reproductor");
      reproductor.addEventListener("click", expandir);
    }
    function expandir() {
      if (!document.webkitFullscreenElement) {
        reproductor.webkitRequestFullscreen();
        video.play();
      } else {
        document.webkitExitFullscreen();
        video.pause();
      }
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section id="reproductor">
    <video id="medio" width="720" height="400" poster="poster.jpg">
      <source src="trailer.mp4">
      <source src="trailer.ogg">
    </video>
  </section>
```

```
</body>
</html>
```

Listado 10-2: *Llevando a cualquier elemento a pantalla completa*

En el Listado 10-2, llevamos el elemento **<section>** y su contenido a pantalla completa. La función **expandir()** se modifica para poder activar y desactivar el modo de pantalla completa para este elemento. Como en el ejemplo anterior, el vídeo se reproduce cuando está en modo de pantalla completa, pero ahora se pausa cuando se cancela el modo.

Estas mejoras son insuficientes para transformar nuestro reproductor en una aplicación de pantalla completa. En el modo de pantalla completa, el nuevo contenedor del elemento es la pantalla, pero las dimensiones originales y los estilos de los elementos **<section>** y **<video>** no se modifican. Al usar la seudoclase **:full-screen**, cambiamos los valores de las propiedades **width** y **height** de estos elementos al 100 %, con lo que se igualan las dimensiones del contenedor. Ahora los elementos ocupan la pantalla completa y se adaptan efectivamente a este modo.

Hágalo usted mismo: cree un nuevo archivo HTML para probar los ejemplos de este capítulo. Una vez que el documento se abre en el navegador, haga clic en el vídeo para activar el modo de pantalla completa y haga clic de nuevo para desactivarlo.

Capítulo 11

API Canvas

11.1 Gráficos

En la introducción de este libro, hemos hablado sobre cómo HTML5 reemplaza tecnologías complementarias como Flash. Había al menos dos temas importantes que considerar para lograr que la Web se independice de tecnologías de terceros: el procesamiento de vídeo y la generación de gráficos. El elemento **<video>** y las API estudiadas en capítulos anteriores cubren este aspecto de forma eficiente, pero no contribuyen con la parte gráfica. Para solucionar este aspecto, los navegadores incluyen la API Canvas. Esta API nos permite dibujar, presentar gráficos, animar y procesar imágenes y texto, y trabaja junto con el resto de las API para crear aplicaciones completas e incluso videojuegos en 2D y 3D para la Web.

El lienzo

La API Canvas solo puede dibujar en el área del documento que se ha asignado para ese propósito. Para definir esa área, HTML incluye el siguiente elemento.

<canvas>—Este elemento crea un área para dibujo. Debe incluir los atributos **width** y **height** para determinar las dimensiones del área.

El elemento **<canvas>** genera un espacio rectangular vacío en la página en el que se mostrará el resultado de los métodos provistos por la API. El elemento produce un espacio en blanco, como un elemento **<div>** vacío, pero para un propósito completamente diferente. El siguiente ejemplo demuestra cómo incluir este elemento en nuestro documento.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Canvas</title>
  <script src="canvas.js"></script>
</head>
<body>
  <section id="cajacanvas">
    <canvas id="canvas" width="500" height="300"></canvas>
  </section>
</body>
</html>
```

Listado 11-1: *Incluyendo el elemento* <canvas>

El contexto

El propósito del elemento **<canvas>** es crear una caja vacía en la pantalla. Para dibujar algo en el lienzo, tenemos que crear un contexto con el siguiente método.

getContext(tipo**)**—Este método genera un contexto de dibujo en el lienzo. Acepta dos valores: `2d` (espacio en dos dimensiones) y `webgl` (espacio en tres dimensiones).

El método **getContext()** es el primer método que tenemos que llamar para preparar el elemento **<canvas>** para trabajar. El resto de la API se aplica a través del objeto que nos devuelve.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");
}
window.addEventListener("load", iniciar);
```

Listado 11-2: *Obteniendo el contexto de dibujo para el lienzo*

En el Listado 11-2 se almacena una referencia al elemento **<canvas>** en la variable **elemento** y se obtiene el contexto 2D mediante el método **getContext()**. El contexto de dibujo 2D del lienzo que devuelve este objeto es una cuadrícula de píxeles listados en filas y columnas de arriba abajo y de izquierda a derecha, con su origen (el píxel 0, 0) ubicado en la esquina superior izquierda.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 11-1. Cree un archivo llamado canvas.js y copie todo el código JavaScript incluido desde el Listado 11-2 en su interior. Cada ejemplo de este capítulo es independiente y reemplaza al anterior. Todas las imágenes utilizadas en este capítulo están disponibles en nuestro sitio web.

Lo básico: actualmente, el contexto `2d` se encuentra disponible en todos los navegadores compatibles con HTML5, mientras que **webgl** solo se aplica en navegadores que han implementado y activado la librería WebGL para la generación de gráficos en 3D. Estudiaremos WebGL en el próximo capítulo.

11.2 Dibujando

Después de preparar el elemento **<canvas>** y su contexto, finalmente podemos comenzar a crear y manipular gráficos. La lista de herramientas provistas por la API con este propósito es extensa, lo que permite la generación de múltiples objetos y efectos, desde formas simples hasta texto, sombras o transformaciones complejas. En esta sección del capítulo, vamos a estudiar estos métodos uno por uno.

Rectángulos

Generalmente, los desarrolladores deben preparar la figura a dibujar antes de enviarla al contexto (como veremos pronto), pero la API incluye algunos métodos que nos permiten dibujar directamente en el lienzo.

fillRect(x, y, ancho, altura**)**—Este método dibuja un rectángulo sólido. La esquina superior izquierda se ubica en la posición especificada por los atributos **x** e **y**. Los atributos **ancho** y **altura** declaran el tamaño del rectángulo.

strokeRect(x, y, ancho, altura**)**—Este método es similar al método anterior, pero dibuja un rectángulo vacío (solo el contorno).

clearRect(x, y, ancho, altura**)**—Este método se usa para extraer píxeles del área especificada por sus atributos. Es como un borrador rectangular.

Dibujar rectángulos es sencillo; solo tenemos que llamar al método en el contexto y la figura se muestra en el lienzo de inmediato.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.strokeRect(100, 100, 120, 120);
  canvas.fillRect(110, 110, 100, 100);
  canvas.clearRect(120, 120, 80, 80);
}
window.addEventListener("load", iniciar);
```

Listado 11-3: *Dibujando rectángulos*

En el ejemplo del Listado 11-3, el contexto se asigna a la variable **`canvas`** y desde esta referencia llamamos a los métodos de dibujo. El primer método, **`strokeRect(100, 100, 120, 120)`**, dibuja un cuadrado vacío con la esquina superior izquierda en la ubicación `100,100` y un tamaño de 120 píxeles. El segundo método, **`fillRect(110, 110, 100, 100)`**, dibuja un cuadrado sólido, esta vez comenzando en la posición `110,110` del lienzo. Y finalmente, con el último método, **`clearRect(120, 120, 80, 80)`**, se extrae un cuadrado de 80 píxeles del centro del cuadrado anterior.

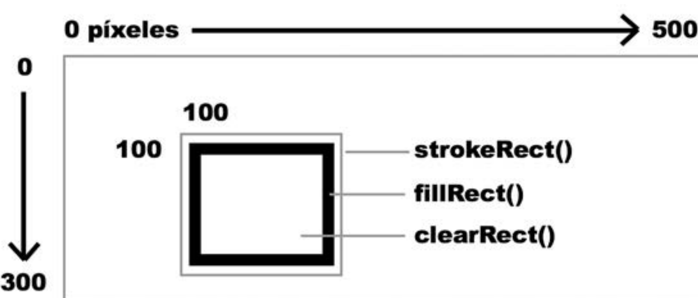

Figura 11-1: *Rectángulos en el lienzo*

La Figura 11-1 es una representación de lo que veremos después de ejecutar el código del Listado 11-3. El elemento **`<canvas>`** se presenta como una cuadrícula de píxeles con su origen en la esquina superior izquierda y un tamaño especificado por sus atributos. Los rectángulos se dibujan en el lienzo en la posición declarada por los atributos **`x`** e **`y`**, y uno sobre otro de acuerdo con el orden del código. El primero en aparecer en el código se dibuja primero, el segundo se dibuja sobre este, y así sucesivamente (existe una propiedad que nos permite determinar cómo se dibujan las figuras, pero la estudiaremos más adelante).

Colores

Hasta el momento hemos usado el color por defecto, el negro, pero podemos especificar el color que queremos mediante la sintaxis de CSS y las siguientes propiedades.

strokeStyle—Esta propiedad declara el color de las líneas de la figura.

fillStyle—Esta propiedad declara el color del interior de la figura.

globalAlpha—Esta propiedad no es para establecer el color sino la transparencia. La propiedad declara transparencia para todas las figuras dibujadas en el lienzo. Los valores posibles oscilan entre 0.0 (completamente opaco) y 1.0 (completamente transparente).

Los colores se definen con los mismos valores que usamos en CSS entre comillas.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.fillStyle = "#000099";
  canvas.strokeStyle = "#990000";

  canvas.strokeRect(100, 100, 120, 120);
  canvas.fillRect(110, 110, 100, 100);
  canvas.clearRect(120, 120, 80, 80);
}
window.addEventListener("load", iniciar);
```

Listado 11-4: *Cambiando colores*

Los colores del Listado 11-4 se declaran usando números hexadecimales, pero también podemos usar funciones como **`rgb()`** e incluso declarar transparencia con la función **`rgba()`**. Estas funciones también se tienen que declarar entre comillas, como en **`strokeStyle = "rgba(255, 165, 0, 1)"`**. Cuando se especifica un nuevo color, este se convierte en el color por defecto para el resto de los dibujos.

Gradientes

Los gradientes son una parte esencial de toda aplicación de dibujo, y la API Canvas no es una excepción. Al igual que en CSS, los gradientes en el lienzo pueden ser lineales o radiales, y podemos indicar límites para combinar colores.

createLinearGradient(x1, y1, x2, y2**)**—Este método crea un objeto que representa un gradiente lineal que podemos aplicar al lienzo.

createRadialGradient(x1, y1, r1, x2, y2, r2**)**—Este método crea un objeto que representa un gradiente radial que se aplica al lienzo usando dos círculos. Los valores representan la posición del centro de cada círculo y sus radios.

addColorStop(posición, color**)**—Este método especifica los colores usados para crear el gradiente. El atributo **posición** es un valor entre 0.0 y 1.0 que determina dónde comienza la degradación del color.

El siguiente ejemplo ilustra cómo aplicar un gradiente lineal a nuestro lienzo.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  var gradiente = canvas.createLinearGradient(0, 0, 10, 100);
  gradiente.addColorStop(0.5, "#00AAFF");
  gradiente.addColorStop(1, "#000000");
  canvas.fillStyle = gradiente;

  canvas.fillRect(10, 10, 100, 100);
  canvas.fillRect(150, 10, 200, 100);
}
window.addEventListener("load", iniciar);
```

Listado 11-5: *Aplicando un gradiente lineal al lienzo*

En el Listado 11-5 hemos creado el objeto del gradiente desde la posición **0,0** a **10,100**, lo que genera una leve inclinación a la izquierda. Los colores se declaran mediante métodos **addColorStop()** y el gradiente final se aplica con la propiedad **fillStyle**, como lo haríamos con un color.

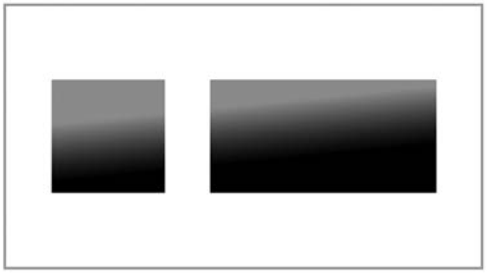

Figura 11-2: *Gradiente lineal para el lienzo*

Las posiciones del gradiente son relativas al lienzo, no a las figuras que queremos afectar. Como resultado, si movemos los rectángulos a una nueva posición en el lienzo, el gradiente de estos rectángulos cambia.

Hágalo usted mismo: el gradiente radial es similar al de CSS. Intente reemplazar el gradiente lineal en el código del Listado 11-5 por un gradiente radial usando una instrucción como **createRadialGradient(0, 0, 30, 0, 0, 300)**. También puede experimentar con la ubicación de los rectángulos para ver cómo se aplica el gradiente.

Trazados

Los métodos estudiados hasta el momento dibujan directamente en el lienzo, pero este no es el procedimiento estándar. Cuando tenemos que dibujar figuras complejas, primero debemos procesar las figuras e imágenes y luego enviar el resultado al contexto para que se dibuja. Para este propósito, la API Canvas introduce varios métodos que nos permiten generar trazados.

Un trazado es como un mapa que el lápiz tiene que seguir. Una vez que se determina el trazado, se tiene que enviar al contexto para ser dibujado. El trazado puede incluir diferentes tipos de líneas (como líneas rectas, arcos, rectángulos y otras) para crear figuras complejas. Los siguientes son los métodos necesarios para iniciar y cerrar un trazado.

beginPath()—Este método inicia un nuevo trazado.

closePath()—Este método cierra el trazado, generando una línea recta desde el último punto hasta el punto inicial. Se puede omitir cuando queremos crear un trazado abierto o cuando usamos el método **`fill()`** para dibujar el trazado.

También contamos con tres métodos para dibujar el trazado en el lienzo.

stroke()—Este método dibuja el trazado como un contorno (sin relleno).

fill()—Este método dibuja el trazado como una figura sólida. Cuando usamos este método, no necesitamos cerrar el trazado con **`closePath()`**, se cierra automáticamente con una línea recta desde el último punto hasta el punto inicial.

clip()—Este método declara una nueva área de recorte para el contexto. Cuando el contexto se inicializa, el área de recorte es todo el área que ocupa el lienzo. El método **`clip()`** cambia el área de recorte a una nueva forma, creando una máscara. Todo lo que queda fuera de la máscara no se dibuja.

Cada vez que queremos crear un trazado, tenemos que llamar al método **`beginPath()`**, como en el siguiente ejemplo.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.beginPath();
  // here goes the path
  canvas.stroke();
}
window.addEventListener("load", iniciar);
```

Listado 11-6: *Iniciando y cerrando un trazado*

El código del Listado 11-6 no crea ninguna figura, solo inicia el trazado y lo dibuja con el método **`stroke()`**, pero no se dibuja nada porque aún no definimos el trazado. Los siguientes son los métodos disponibles para declarar el trazado y crear la figura.

moveTo(x, y**)**—Este método mueve el lápiz a una nueva posición. Nos permite comenzar o continuar el trazado desde puntos diferentes de la cuadrícula, evitando líneas continuas.

lineTo(x, y**)**—Este método genera una línea recta desde la posición actual del lápiz a la declarada por los atributos **x** e **y**.

rect(x, y, ancho, altura**)**—Este método genera un rectángulo. A diferencia de los métodos estudiados anteriormente, el rectángulo generado por este método es parte del trazado (no se dibuja directamente en el lienzo). Los atributos cumplen la misma función que los demás métodos.

arc(x, y, radio, ángulo_inicial, ángulo_final, dirección**)**—Este método genera un arco o un círculo con el centro en las coordenadas indicadas por **x** e **y**, y con un radio y ángulos declarados por el resto de los atributos. El último valor se tiene que declarar como `true` o `false` para indicar la dirección (en dirección opuesta a las agujas del reloj o hacia el mismo lado, respectivamente).

quadraticCurveTo(cpx, cpy, x, y**)**—Este método genera una curva Bézier cuadrática que comienza en la posición actual del lápiz y finaliza en la posición que declaran los atributos `x` e `y`. Los atributos **cpx** y **cpy** definen el punto de control que le da forma a la curva.

bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y**)**—Este método es similar al anterior pero agrega dos atributos más para generar una curva Bézier cúbica. El método genera dos puntos de control en la cuadrícula declarados por los valores **cp1x**, **cp1y**, **cp2x** y **cp2y** para dar forma a la curva.

El siguiente código genera un trazado sencilla que ilustra cómo trabajar con estos métodos.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.beginPath();
  canvas.moveTo(100, 100);
  canvas.lineTo(200, 200);
  canvas.lineTo(100, 200);
  canvas.stroke();
}
window.addEventListener("load", iniciar);
```

Listado 11-7: *Creando un trazado*

Siempre se recomienda declarar la posición inicial del lápiz inmediatamente después de iniciar el trazado. En el código del Listado 11-7, primero movemos el lápiz a la posición `100,100` y luego generamos una línea desde este punto hasta el punto `200,200`. La posición del lápiz ahora es `200,200`, y la siguiente línea se dibuja desde ese punto hasta el punto `100,200`. Finalmente, el trazado se dibuja como un contorno con el método `stroke()`.

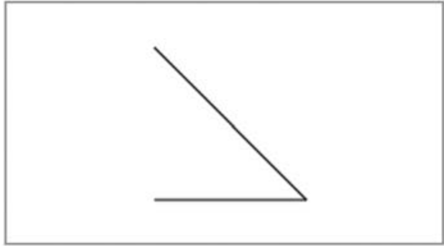

Figura 11-3: *Trazado abierto*

La Figura 11-3 muestra una representación del triángulo abierto generado con el código del Listado 11-7. Este triángulo se puede cerrar o incluso rellenar con diferentes métodos, tal como muestran los siguientes ejemplos.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.beginPath();
  canvas.moveTo(100, 100);
  canvas.lineTo(200, 200);
  canvas.lineTo(100, 200);

  canvas.closePath();
  canvas.stroke();
}
window.addEventListener("load", iniciar);
```

Listado 11-8: *Completando el triángulo*

El método **closePath()** agrega una línea recta al trazado, desde el punto final al punto inicial, cerrando la figura.

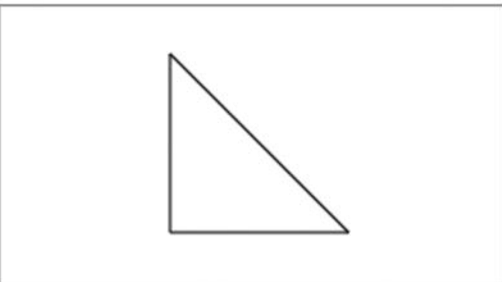

Figura 11-4: *Trazado cerrado*

Usando el método **stroke()** al final de nuestro trazado, dibujamos un triángulo vacío en el lienzo. Si queremos crear un triángulo sólido, tenemos que usar el método **fill()**.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.beginPath();
  canvas.moveTo(100, 100);
  canvas.lineTo(200, 200);
  canvas.lineTo(100, 200);
  canvas.fill();
}
window.addEventListener("load", iniciar);
```

Listado 11-9: *Dibujando un triángulo sólido*

Ahora, la figura de la pantalla es un triángulo sólido. El método **fill()** cierra el trazado automáticamente y, por lo tanto, ya no tenemos que usar el método **closePath()**.

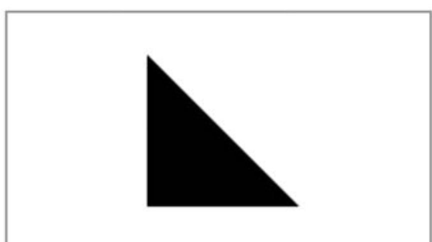

Figura 11-5: *Triángulo sólido*

Uno de los métodos que hemos introducido antes para dibujar un trazado en el lienzo ha sido **`clip()`**. Este método no dibuja nada, sino que crea una máscara con la forma del trazado para seleccionar lo que se dibujará y lo que no. Todo lo que cae fuera de la máscara no se dibujará.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.beginPath();
  canvas.moveTo(100, 100);
  canvas.lineTo(200, 200);
  canvas.lineTo(100, 200);
  canvas.clip();
  canvas.beginPath();
  for (var f = 0; f < 300; f = f + 10) {
    canvas.moveTo(0, f);
    canvas.lineTo(500, f);
  }
  canvas.stroke();
}
window.addEventListener("load", iniciar);
```

Listado 11-10: *Usando un triángulo como máscara*

Para mostrar cómo trabaja el método **`clip()`**, en el Listado 11-10 hemos creado un bucle **`for`** que genera líneas horizontales cada 10 píxeles. Las líneas van desde el lado izquierdo al lado derecho del lienzo, pero solo se dibujan las partes de las líneas que quedan dentro del triángulo.

Figura 11-6: *Área de recorte*

Ahora que sabemos cómo dibujar trazados, es hora de analizar otras alternativas que tenemos para crear figuras. Hasta el momento, hemos aprendido cómo generar líneas rectas y figuras cuadradas. Para crear figuras circulares, la API ofrece tres métodos: **`arc()`**,

quadraticCurveTo() y **bezierCurveTo()**. El primero es relativamente sencillo de usar y puede generar círculos completos o parciales, según muestra el siguiente ejemplo.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.beginPath();
  canvas.arc(100, 100, 50, 0, Math.PI * 2, false);
  canvas.stroke();
}
window.addEventListener("load", iniciar);
```

Listado 11-11: *Dibujando círculos con el método* `arc()`

En el método **arc()** del Listado 11-11 usamos el valor PI del objeto **Math** para especificar el ángulo. Esto es necesario porque este método usa radianes en lugar de grados. En radianes, el valor de PI representa 180 grados, y en consecuencia la fórmula **PI × 2** multiplica PI por 2, con lo que se obtienen 360 grados.

Este ejemplo genera un arco con el centro en el punto **100,100** y un radio de 50 píxeles, comenzando a los **0** grados y terminando a los **Math.PI * 2** grados, lo cual representa un círculo completo.

Si necesitamos calcular el valor en radianes a partir de grados, tenemos que usar la fórmula: **Math.PI / 180 × grados**, como en el siguiente ejemplo.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  var radianes = Math.PI / 180 * 45;

  canvas.beginPath();
  canvas.arc(100, 100, 50, 0, radianes, false);
  canvas.stroke();
}
window.addEventListener("load", iniciar);
```

Listado 11-12: *Dibujando un arco de 45 grados*

Con el código del Listado 11-12 obtenemos un arco que cubre 45 grados de un círculo, pero si cambiamos el valor de la dirección a **true** en el método **arc()**, el arco se genera desde 0 grados a 315 y crea un círculo abierto, tal como ilustra la Figura 11-7.

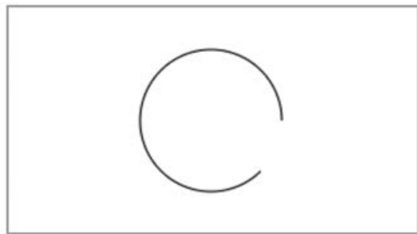

Figura 11-7: *Semicírculo con el método* `arc()`

Lo básico: si continúa trabajando con este trazado, el punto de inicio actual será el final del arco. Si no quiere comenzar en el final del arco, tendrá que usar el método **`moveTo()`** para cambiar la posición del lápiz. Sin embargo, si la siguiente figura es otro arco, debe recordar que el método **`moveTo()`** mueve el lápiz virtual al punto en el que el círculo se comienza a dibujar, no al centro del círculo.

Además de **`arc()`**, tenemos otros dos métodos para dibujar curvas complejas. El método **`quadraticCurveTo()`** genera una curva Bézier cuadrática, y el método **`bezierCurveTo()`** genera una curva Bézier cúbica. La diferencia entre estos métodos es que el primero tiene solo un punto de control mientras que el segundo tiene dos, por lo que genera diferentes curvas.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.beginPath();
  canvas.moveTo(50, 50);
  canvas.quadraticCurveTo(100, 125, 50, 200);

  canvas.moveTo(250, 50);
  canvas.bezierCurveTo(200, 125, 300, 125, 250, 200);
  canvas.stroke();
}
window.addEventListener("load", iniciar);
```

Listado 11-13: *Creando curvas complejas*

Para crear una curva cuadrática en este ejemplo, movemos el lápiz virtual a la posición `50,50` y finalizamos la curva en la posición `50,200`. El punto de control de esta curva se encuentra en la posición `100,125`. La curva cúbica generada por el método **`bezierCurveTo()`** es un poco más complicada. Tenemos dos puntos de control para esta curva, el primero en la posición `200,125` y el segundo en la posición `300,125`. El resultado se muestra en la Figura 11-8.

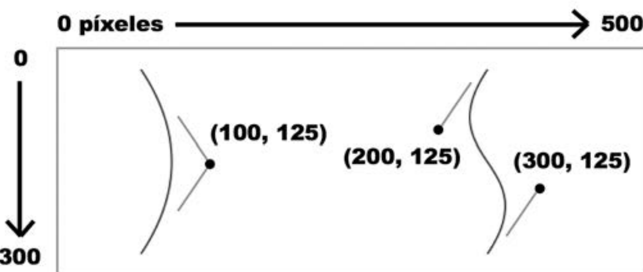

Figura 11-8: *Representación de curvas Bézier y sus puntos de control en el lienzo*

Los valores de la Figura 11-8 indican la posición de los puntos de control de ambas curvas. Si movemos estos puntos de control, cambiamos la forma de la curva.

Hágalo usted mismo: puede agregar todas las curvas que necesite para construir la figura. Intente cambiar los valores de los puntos de control en el Listado 11-13 para ver cómo afectan a las curvas. Construya formas más complejas combinando curvas y líneas para entender cómo se crea un trazado.

Líneas

Hasta este momento hemos usado el mismo estilo de línea para dibujar en el lienzo. Se pueden manipular el ancho, los extremos y otros aspectos de la línea para obtener el tipo de línea exacto que necesitamos para nuestros dibujos. Para este propósito, la API incluye cuatro propiedades.

lineWidth—Esta propiedad determina el grosor de la línea. Por defecto, el valor es 1.0.

lineCap—Esta propiedad determina la forma de los extremos de la línea. Los valores disponibles son **`butt`**, **`round`**, y **`square`**.

lineJoin—Esta propiedad determina la forma de la conexión entre dos líneas. Los valores disponibles son **`round`**, **`bevel`** y **`miter`**.

miterLimit—Esta propiedad determina la distancia a la que se extenderán las conexiones de las líneas cuando se declara la propiedad **`lineJoin`** con el valor **`miter`**.

Estas propiedades afectan al trazado completo. Cada vez que queremos cambiar las características de las líneas, tenemos que crear un nuevo trazado con nuevos valores.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.beginPath();
  canvas.arc(200, 150, 50, 0, Math.PI * 2, false);
  canvas.stroke();

  canvas.lineWidth = 10;
  canvas.lineCap = "round";

  canvas.beginPath();
  canvas.moveTo(230, 150);
  canvas.arc(200, 150, 30, 0, Math.PI, false);
  canvas.stroke();
  canvas.lineWidth = 5;
  canvas.lineJoin = "miter";
  canvas.beginPath();
  canvas.moveTo(195, 135);
  canvas.lineTo(215, 155);
  canvas.lineTo(195, 155);
  canvas.stroke();
}
window.addEventListener("load", iniciar);
```

Listado 11-14: *Probando las propiedades para las líneas*

Hemos comenzado el dibujo del ejemplo del Listado 11-14 definiendo un trazado para un círculo completo con las propiedades por defecto. Luego, usando la propiedad **`lineWith`**,

cambiamos el grosor de la línea a `10` y declaramos la propiedad `lineCap` como `round`. Esto hace que el trazado sea grueso y con extremos redondeados, lo que nos ayudará a crear una boca sonriente. Para crear el trazado, movemos el lápiz a la posición `230,150` y luego generamos un semicírculo. Finalmente, agregamos otro trazado con dos líneas para formar una figura similar a una nariz. Las líneas para este trazado tienen un grosor de `5` y se unen con la propiedad `lineJoin` y el valor `miter`. Esta propiedad hace que la nariz sea puntiaguda y los extremos de las esquinas se expandan hasta que alcanzan un punto en común.

Figura 11-9: *Diferentes tipos de línea*

Hágalo usted mismo: experimente cambiando las líneas para la nariz y modifique la propiedad `miterLimit` (por ejemplo, `miterLimit = 2`). Cambie el valor de la propiedad `lineJoin` a `round` o `bevel`. También puede modificar la forma de la boca probando diferentes valores para la propiedad `lineCap`.

Texto

Escribir texto en el lienzo es tan sencillo como definir unas pocas propiedades y llamar a los métodos apropiados. Contamos con tres propiedades para configurar el texto.

font—Esta propiedad declara el tipo de letra que se va a usar para el texto. Acepta los mismos valores que la propiedad `font` de CSS.

textAlign—Esta propiedad alinea el texto horizontalmente. Los valores disponibles son `start`, `end`, `left`, `right` y `center`.

textBaseline—Esta propiedad es usa para el alineamiento vertical. Declara diferentes posiciones para el texto (incluido texto Unicode). Los valores disponibles son `top`, `hanging`, `middle`, `alphabetic`, `ideographic`, y `bottom`.

Los siguientes son los métodos disponibles para dibujar el texto.

strokeText(texto, x, y**)**—Este método es similar al método para trazados. Dibuja el texto especificado como un contorno en las posiciones **x** e **y**. También puede incluir un cuarto valor para declarar el tamaño máximo. Si el texto es más largo que este valor, se reducirá para que quepa dentro de ese espacio.

fillText(texto, x, y**)**—Este método es similar al método previo, pero dibuja texto sólido (con relleno).

El siguiente ejemplo demuestra cómo dibujar un texto sencillo con un tipo de letra personalizado y en una posición específica en el lienzo.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.font = "bold 24px verdana, sans-serif";
  canvas.textAlign = "start";
  canvas.fillText("Mi Mensaje", 100, 100);
}
window.addEventListener("load", iniciar);
```

Listado 11-15: *Dibujando texto*

Al igual que en CSS, la propiedad **`font`** puede recibir varios valores al mismo tiempo. En este ejemplo, usamos esta propiedad para definir el tipo de letra y su tamaño, y luego declaramos la propiedad **`textAlign`** para especificar que el texto se debe comenzar a dibujar en la posición **`100,100`** (Si el valor de esta propiedad fuera **`end`**, por ejemplo, el texto hubiese terminado en la posición **`100,100`**). Finalmente, el método **`fillText`** dibuja un texto sólido en el lienzo.

Además de los métodos ya mencionados, la API provee otro método importante para trabajar con texto.

measureText(texto**)**—Este método devuelve información acerca del tamaño del texto entre paréntesis.

El método **`measureText()`** puede resultar útil para calcular posiciones o incluso colisiones en animaciones, y también para combinar texto con otras figuras en el lienzo, como hacemos en el siguiente ejemplo.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.font = "bold 24px verdana, sans-serif";
  canvas.textAlign = "start";
  canvas.textBaseline = "bottom";
  canvas.fillText("Mi Mensaje", 100, 124);

  var tamano = canvas.measureText("Mi Mensaje");
  canvas.strokeRect(100, 100, tamano.width, 24);
}
window.addEventListener("load", iniciar);
```

Listado 11-16: *Midiendo texto*

En este ejemplo, hemos comenzado con el mismo código del Listado 11-15, pero esta vez la propiedad **`textBaseline`** es declarada con el valor **`bottom`**, lo que significa que la parte más baja del texto se ubicará en la posición **`124`**. De esta manera sabemos la posición vertical exacta del texto en el lienzo. A continuación, usando el método **`measureText()`** y la propiedad **`width`**, obtenemos el tamaño horizontal del texto. Con todas las medidas tomadas, ahora podemos dibujar un rectángulo que rodee al texto.

Figura 11-10: *Trabajando con texto*

Hágalo usted mismo: use el código del Listado 11-16 para probar diferentes valores para las propiedades `textAlign` y `textBaseline`. Use el rectángulo como referencia para ver cómo trabajan. Escriba diferentes textos para ver cómo el rectángulo se adapta automáticamente a cada tamaño.

IMPORTANTE: el método `measureText()` devuelve un objeto con varias propiedades. La propiedad `width` usada en nuestro ejemplo es solo una de ellas. Para obtener más información, lea la especificación de la API Canvas. El enlace se encuentra disponible en nuestro sitio web.

Sombras

Las sombras son, por supuesto, también una parte importante de la API Canvas. Podemos generar sombras para cada trazado, incluidos textos. La API incluye cuatro propiedades para definir una sombra.

shadowColor—Esta propiedad declara el color de la sombra usando sintaxis CSS.

shadowOffsetX—Esta propiedad determina la distancia a la que estará la sombra del objeto en la dirección horizontal.

shadowOffsetY—Esta propiedad determina la distancia a la que estará la sombra del objeto en la dirección vertical.

shadowBlur—Esta propiedad produce un efecto de difuminado para la sombra.

Las propiedades se deben definir antes de que se dibuje el trazado en el lienzo. El siguiente ejemplo genera una sombra para un texto.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.shadowColor = "rgba(0, 0, 0, 0.5)";
  canvas.shadowOffsetX = 4;
  canvas.shadowOffsetY = 4;
  canvas.shadowBlur = 5;

  canvas.font = "bold 50px verdana, sans-serif";
  canvas.fillText("Mi Mensaje", 100, 100);
}
window.addEventListener("load", iniciar);
```

Listado 11-17: *Aplicando sombras a textos*

La sombra creada en el Listado 11-17 usa la función **`rgba()`** para obtener un color negro transparente. La sombra se desplaza 4 píxeles y tiene un difuminado de 5 píxeles.

Figura 11-11: *Texto con sombra*

Transformaciones

El lienzo se puede transformar, lo que afectará al dibujo de las figuras. Los siguientes son los métodos disponibles para realizar estas operaciones.

translate(x, y**)**—Este método se utiliza para mover el origen del lienzo.

rotate(ángulo**)**—Este método rota el lienzo alrededor del origen en los radianes especificados por el atributo.

scale(x, y**)**—Este método incrementa o disminuye las unidades en el lienzo, reduciendo o expandiendo todo lo que haya dibujado en el mismo. La escala se puede cambiar de forma independiente en los ejes horizontal y vertical usando los atributos **x** e **y**. Los valores pueden ser negativos, lo cual produce un efecto espejo. Por defecto, los valores se declaran como 1.0.

transform(m1, m2, m3, m4, dx, dy**)**—Este método aplica una nueva matriz sobre la actual para modificar las propiedades del lienzo.

setTransform(m1, m2, m3, m4, dx, dy**)**—Este método reinicia la matriz de transformación actual y declara una nueva a partir de los valores provistos por los atributos.

Todo lienzo tiene un punto 0, 0 (el origen) localizado en la esquina superior izquierda, y sus valores se incrementan en toda dirección dentro del lienzo (los valores negativos determinan posiciones fuera del lienzo). El método **`translate()`** nos permite mover el origen a una nueva posición para usarlo como referencia para nuestros dibujos. En el siguiente ejemplo, movemos el origen varias veces para cambiar la posición de un texto.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.font = "bold 20px verdana, sans-serif";
  canvas.fillText("PRUEBA", 50, 20);

  canvas.translate(50, 70);
  canvas.rotate(Math.PI / 180 * 45);
  canvas.fillText("PRUEBA", 0, 0);

  canvas.rotate(-Math.PI / 180 * 45);
  canvas.translate(0, 100);
```

```
  canvas.scale(2, 2);
  canvas.fillText("PRUEBA", 0, 0);
}
window.addEventListener("load", iniciar);
```

Listado 11-18: *Traduciendo, rotando y escalando*

En el Listado 11-18 hemos aplicado los métodos `translate()`, `rotate()` y `scale()` al mismo texto. Primero, dibujamos un texto en el lienzo con el estado del lienzo por defecto. El texto aparece en la posición `50,20` con un tamaño de 20 píxeles. Usando `translate()`, el origen del lienzo se mueve a la posición `50,70`, y el lienzo completo rota 45 grados con el método `rotate()`. En consecuencia, el siguiente texto se dibuja en el nuevo origen y con una inclinación de 45 grados. La transformaciones aplicadas al lienzo son ahora los valores por defecto, por lo que para probar el método `scale()`, rotamos el lienzo de nuevo unos 45 a su posición original y desplazamos el origen hacia abajo unos 100 píxeles. Finalmente, se duplica la escala del lienzo y se dibuja otro texto, esta vez al doble del tamaño del texto original.

Figura 11-12: *Aplicando transformaciones*

IMPORTANTE: cada transformación es acumulativa. Si realizamos dos transformaciones usando `scale()`, por ejemplo, el segundo método aplicará la escala usando el estado actual. Un método `scale(2,2)` después de otro método `scale(2,2)` cuadruplicará la escala del lienzo. Esto se aplica a cualquier transformación, incluidos los métodos que transforman la matriz, como veremos a continuación.

El lienzo tiene una matriz de valores que definen sus propiedades. Modificando esta matriz, podemos cambiar las características del lienzo. La API ofrece dos métodos con este propósito: `transform()` y `setTransform()`.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.transform(3, 0, 0, 1, 0, 0);
  canvas.font = "bold 20px verdana, sans-serif";
  canvas.fillText("PRUEBA", 20, 20);

  canvas.transform(1, 0, 0, 10, 0, 0);
  canvas.font = "bold 20px verdana, sans-serif";
  canvas.fillText("PRUEBA", 20, 20);
}
window.addEventListener("load", iniciar);
```

Listado 11-19: *Transformaciones acumulativas de la matriz*

Como en el ejemplo anterior, en el Listado 11-19 aplicamos métodos de transformación al mismo texto para comparar efectos. Los valores por defecto de la matriz del lienzo son `1, 0, 0, 1, 0, 0`. Si cambiamos el primer valor a `3` en la primera transformación, estiramos el lienzo en el eje horizontal. El texto dibujado después de esta transformación es más ancho que el texto dibujado en condiciones normales.

Con la siguiente transformación en el código, el lienzo se estira verticalmente cambiando el cuarto valor a `10` y preservando el resto. El resultado se ilustra a continuación.

Figura 11-13: *Modificando la matriz de transformación*

Algo que se debe tener en cuenta es que las transformaciones se aplican a la matriz establecida por la transformación anterior, por lo que el segundo texto que muestra el código del Listado 11-19 se estira horizontalmente y verticalmente, como ilustra la Figura 11-13. Para reiniciar la matriz y declarar nuevos valores de transformación, podemos usar el método **`setTransform()`**.

Estado

La acumulación de transformaciones hace que sea difícil volver a una estado previo. En el código del Listado 11-18, por ejemplo, hemos tenido que recordar el valor de la rotación para poder realizar una nueva rotación con la que deshacer las transformaciones anteriores. Considerando esta situación, la API Canvas facilita dos métodos con los que podemos grabar y recuperar el estado del lienzo.

save()—Este método graba el estado del lienzo, incluidas las transformaciones aplicadas, los valores de las propiedades de estilos y el área de recorte actual (el área creada por el método **`clip()`**).

restore()—Este método restaura el último estado del lienzo grabado.

Primero, tenemos que almacenar el estado que queremos preservar con el método **`save()`** y luego recuperarlo con el método **`restore()`**. Cualquier cambio realizado en el lienzo entre medio no afectará a los dibujos una vez que se restaura el estado.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.save();
  canvas.translate(50, 70);
  canvas.font = "bold 20px verdana, sans-serif";
  canvas.fillText("PRUEBA1", 0, 30);
  canvas.restore();
```

```
  canvas.fillText("PRUEBA2", 0, 30);
}
window.addEventListener("load", iniciar);
```

Listado 11-20: *Grabando y restaurando el estado del lienzo*

Después de ejecutar el código JavaScript del Listado 11-20, el texto "PRUEBA1" se dibuja en letras grandes en el centro del lienzo y el texto "PRUEBA2" en un tamaño de letra más pequeño cerca del origen. Lo que hacemos en este ejemplo es grabar el estado del lienzo por defecto y luego declarar una nueva posición para el origen y los estilos del texto. Antes de dibujar el segundo texto en el lienzo, se restaura el estado original. Como consecuencia, el segundo texto se muestra con los estilos por defecto, no con los que se han declarado para el primero.

Independientemente de las transformaciones realizadas en el lienzo, el estado volverá exactamente a la condición anterior cuando se llame al método **`restore()`**.

La propiedad GlobalCompositeOperation

Cuando hemos hablado acerca de los trazados, hemos comentado que existe una propiedad con la que podemos determinar cómo se posiciona y se combina una figura con figuras previas en el lienzo. Esa propiedad es **`globalCompositeOperation`** y su valor por defecto es **`source-over`**, lo cual significa que la nueva figura se dibujará sobre las que ya existen en el lienzo. Existen otros 11 valores disponibles.

source-in—Solo se dibuja la parte de la nueva figura que se superpone a la figura en el lienzo. El resto de la nueva figura y el resto de la figura en el lienzo se vuelven transparentes.

source-out—Solo se dibuja la parte de la nueva figura que no se superpone a la figura en el lienzo. El resto de la nueva figura y el resto de la figura en el lienzo se vuelven transparentes.

source-atop—Solo se dibuja la parte de la nueva figura que se superpone a la figura en el lienzo. La figura en el lienzo se preserva, pero el resto de la nueva figura se vuelve transparente.

lighter—Se dibujan ambas figuras, pero el color de las partes que se superponen se determina sumando los valores de los colores.

xor—Se dibujan ambas figuras, pero las partes que se superponen se vuelven transparentes.

destination-over—Este valor es el opuesto al valor por defecto (**`source-over`**). La nueva figura se dibuja debajo de las figuras existentes en el lienzo.

destination-in—Se preservan las partes de la figura en el lienzo que se superponen con la nueva figura. El resto, incluida la nueva figura, se vuelven transparentes.

destination-out—Se preservan las partes de la figura en el lienzo que se superponen con la nueva figura. El resto, incluida la nueva figura, se vuelven transparente.

destination-atop—La figura en el lienzo y la nueva figura solo se preservan donde se superponen.

darker—Se dibujan ambas figuras, pero el color de las partes que se superponen se determina restando los valores de los colores.

copy—Solo se dibuja la nueva figura. La figura en el lienzo se vuelve transparente.

Como con la mayoría de las propiedades de la API, primero tenemos que definir la propiedad y luego dibujar el trazado en el lienzo.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  canvas.fillStyle = "#666666";
  canvas.fillRect(50, 100, 300, 80);

  canvas.globalCompositeOperation = "source-atop";

  canvas.fillStyle = "#DDDDDD";
  canvas.font = "bold 60px verdana, sans-serif";
  canvas.textAlign = "center";
  canvas.textBaseline = "middle";
  canvas.fillText("PRUEBA", 200, 100);
}
window.addEventListener("load", iniciar);
```

Listado 11-21: *Probando la propiedad* `globalCompositeOperation`

Solo una representación visual de cada uno de los valores de la propiedad **`globalCompositeOperation`** puede ayudarnos a entender cómo funcionan. Esta es la razón por la que hemos preparado el ejemplo del Listado 11-21. Cuando se ejecuta este código, se dibuja un rectángulo rojo en el medio del lienzo, pero como resultado del valor **`source-atop`**, solo se dibuja en la pantalla la parte del texto que se superpone al rectángulo.

Figura 11-14: *Aplicando* `globalCompositeOperation`

Hágalo usted mismo: reemplace el valor **`source-atop`** con cualquiera de los otros valores disponibles para esta propiedad y compruebe el efecto en su navegador. Pruebe el código en diferentes navegadores.

11.3 Imágenes

La API Canvas sería inútil sin la capacidad de procesar imágenes, pero a pesar de la importancia de las imágenes, solo se requiere un método para dibujarlas en el lienzo. Sin embargo, existen tres versiones disponibles de este método que producen diferentes resultados. Estas son las posibles combinaciones.

drawImage(imagen, x, y**)**—Esta sintaxis se usa para dibujar una imagen en el lienzo en la posición especificada por los atributos **x** e **y**. El primer atributo es una referencia a la imagen, la cual puede ser una referencia a un elemento **<img>**, **<video>** o **<canvas>**.

drawImage(imagen, x, y, ancho, altura**)**—Esta sintaxis nos permite escalar la imagen antes de dibujarla en el lienzo, cambiando su tamaño a los valores de los atributos **ancho** y **altura**.

drawImage(imagen, x1, y1, ancho1, altura1, x2, y2, ancho2, altura2**)**—Esta es la sintaxis más compleja. Incluye dos valores para cada parámetro. El propósito es poder cortar una parte de la imagen y luego dibujarla en el lienzo con un tamaño y posición personalizados. Los atributos **x1** e **y1** declaran la esquina superior izquierda de la parte de la imagen que se cortará, y los atributos **ancho1** y **altura1** indican su tamaño. El resto de los atributos (**x2**, **y2**, **ancho2** y **altura2**) declaran dónde se dibujará la parte de la imagen en el lienzo y su tamaño (el cual puede ser diferente del original).

En cada caso, el primer atributo es siempre una referencia a la imagen o un elemento de medios, incluido otro lienzo. No es posible usar una URL o cargar un archivo desde fuentes externas directamente con este método, primero tenemos que crear un elemento **<img>** para cargar la imagen y luego llamar al método con una referencia a este elemento, como hacemos en el siguiente ejemplo.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  var imagen = document.createElement("img");
  imagen.src = "nieve.jpg";
  imagen.addEventListener("load", function() {
    canvas.drawImage(imagen, 20, 20);
  });
}
window.addEventListener("load", iniciar);
```

Listado 11-22: *Dibujando una imagen*

El código del Listado 11-22 carga una imagen y la dibuja en el lienzo. Debido a que el lienzo solo puede recibir imágenes que ya se han descargado, necesitamos controlar esta situación con el evento **load**. Después de que se crea la imagen con el método **createElement()** y se descarga, se dibuja en la posición **20**, **20** con el método **drawImage()**.

Figura 11-15: *Imagen en el lienzo*

El siguiente ejemplo ilustra cómo redimensionar una imagen agregando más atributos al método **drawImage()**.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  var imagen = document.createElement("img");
  imagen.src = "nieve.jpg";
  imagen.addEventListener("load", function() {
    canvas.drawImage(imagen, 0, 0, elemento.width, elemento.height);
  });
}
window.addEventListener("load", iniciar);
```

Listado 11-23: *Ajustando la imagen al tamaño del lienzo*

En el Listado 11-23, el tamaño de la imagen queda determinado por las propiedades **width** y **height** del elemento **<canvas>**, por lo que la imagen se estira con el método hasta cubrir todo el lienzo.

Figura 11-16: *Imagen estirada para cubrir el lienzo*

El siguiente ejemplo implementa la sintaxis más compleja del método **drawImage()** para extraer un trozo de la imagen original y dibujarlo en el lienzo con un tamaño diferente.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var canvas = elemento.getContext("2d");

  var imagen = document.createElement("img");
  imagen.src = "nieve.jpg";
  imagen.addEventListener("load", function() {
    canvas.drawImage(imagen, 135, 30, 50, 50, 0, 0, 300, 300);
  });
}
window.addEventListener("load", iniciar);
```

Listado 11-24: *Extrayendo, redimensionando y dibujando*

En este ejemplo, hemos tomado un recuadro de la imagen original comenzando en la posición `135,50`, y con un tamaño de `50,50` píxeles. El trozo de imagen se redimensiona a `300,300` píxeles y finalmente se dibuja en el lienzo en la posición `0,0`.

Figura 11-17: *Un trozo de la imagen en el lienzo*

Patrones

Los patrones nos permiten agregar una textura a las figuras usando una imagen. El procedimiento es similar a los gradientes: se crea el patrón y luego se aplica al trazado como un color. El siguiente es el método que se incluye en la API para crear un patrón.

createPattern(imagen, tipo**)**—Este método crea y devuelve un objeto que representa un patrón. El atributo **imagen** es una referencia a la imagen que queremos usar como patrón, y el atributo **tipo** determina su tipo. Los valores disponibles son **`repeat`**, **`repeat-x`**, **`repeat-y`**, y **`no-repeat`**.

Al igual que los gradientes, el objeto que representa al patrón primero se debe crear y luego se debe asignar a la propiedad **`fillStyle`** del lienzo.

```
var canvas, imagen;
function iniciar() {
  var elemento = document.getElementById("canvas");
  canvas = elemento.getContext("2d");
  imagen = document.createElement("img");
  imagen.src = "ladrillos.jpg";
  imagen.addEventListener("load", agregarpatron);
}
function agregarpatron() {
  var patron = canvas.createPattern(imagen, "repeat");
  canvas.fillStyle = patron;
  canvas.fillRect(0, 0, 500, 300);
}
window.addEventListener("load", iniciar);
```

Listado 11-25: *Agregando un patrón a nuestros trazados*

El código del Listado 11-25 crea un rectángulo del tamaño del lienzo y lo rellena con un patrón usando la imagen ladrillos.jpg.

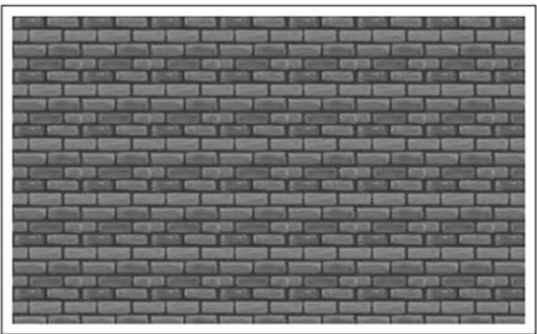

Figura 11-18: *Patrón*

Hágalo usted mismo: actualice su archivo canvas.js con el código del Listado 11-25. Descargue el archivo ladrillos.jpg desde nuestro sitio web y abra el documento del Listado 11-1 en su navegador. Debería ver algo parecido a la Figura 11-18. Experimente con los diferentes valores disponibles para el método **`createPattern()`** y asigne el patrón a otras figuras.

Datos de imagen

Anteriormente, explicamos que **`drawImage()`** era el único método disponible para dibujar una imagen en el lienzo, pero esto no es del todo correcto. También existen métodos para procesar imágenes que se pueden dibujar en el lienzo; sin embargo, estos métodos trabajan con datos, no con imágenes.

getImageData(x, y, ancho, altura**)**—Este método toma un rectángulo del lienzo del tamaño declarado por los atributos y lo convierte en datos. El método devuelve un objeto con las propiedades **`width`**, **`height`** y **`data`**.

putImageData(datosimagen, x, y**)**—Este método convierte los datos declarados por el atributo **datosimagen** en una imagen y la dibuja en el lienzo en la posición especificada por los atributos **x** e **y**. Es lo opuesto de **`getImageData()`**.

Todas las imágenes se pueden describir como una secuencia de números enteros que representan valores RGBA. Hay cuatro valores por cada píxel que definen los colores rojo, verde, azul y alfa (transparencia). Esta información crea un array unidimensional que se puede usar para generar una imagen. La posición de cada valor en el array se calcula con la fórmula **(ancho × 4 × y) + (x × 4) + n**, donde **n** es un índice para los valores del píxel, comenzando por 0. La fórmula para el rojo es **(ancho × 4 × y) + (x × 4) + 0**; para verde es **(ancho × 4 × y) + (x × 4) + 1**; para el azul es **(ancho × 4 × y) + (x × 4) + 2**; y para el valor alfa es **(ancho × 4 × y) + (x × 4) + 3**. El siguiente ejemplo implementa estas fórmulas para crear el negativo de una imagen.

```
var canvas, imagen;
function iniciar() {
  var elemento = document.getElementById("canvas");
  canvas = elemento.getContext("2d");

  imagen = document.createElement("img");
```

```
  imagen.src = "nieve.jpg";
  imagen.addEventListener("load", modimagen);
}
function modimagen() {
  canvas.drawImage(imagen, 0, 0);
  var info = canvas.getImageData(0, 0, 175, 262);
  var pos;
  for (var x = 0; x < 175; x++) {
    for (var y = 0; y < 262; y++) {
      pos = (info.width * 4 * y) + (x * 4);
      info.data[pos] = 255 - info.data[pos];
      info.data[pos+1] = 255 - info.data[pos+1];
      info.data[pos+2] = 255 - info.data[pos+2];
    }
  }
  canvas.putImageData(info, 0, 0);
}
window.addEventListener("load", iniciar);
```

Listado 11-26: *Generando el negativo de una imagen*

IMPORTANTE: estos métodos presentan limitaciones para el acceso de origen cruzado. Los archivos de este ejemplo se tienen que subir a un servidor o un servidor local para que funcionen (incluido el archivo nieve.jpg que puede descargar desde nuestro sitio web). A continuación estudiaremos el acceso de origen cruzado.

En el ejemplo del Listado 11-26, hemos creado una nueva función para procesar la imagen llamada **`modimagen()`**. Esta función dibuja la imagen en el lienzo en la posición **`0,0`** usando el método **`drawImage()`** y luego procesa los datos de la imagen píxel a píxel.

La imagen de nuestro ejemplo tiene 350 píxeles de ancho y 262 píxeles de alto. Usando el método **`getImageData()`** con los valores **`0,0`** para la esquina superior izquierda y **`175,262`** para las dimensiones horizontal y vertical, extraemos la mitad izquierda de la imagen original. Los datos obtenidos se almacenan en la variable **`info`** y luego se procesan para lograr el efecto deseado.

Debido a que cada color se declara mediante un valor entre 0 y 255 (un byte), el valor negativo se obtiene restando 255 al valor original con la fórmula **`color = 255 - color`**. Para llevar a cabo esta tarea por cada píxel de nuestra imagen creamos dos bucles **`for`**, uno para las columnas y otro para las filas. El bucle **`for`** para los valores **`x`** va de 0 a 174 (el ancho de la parte de la imagen que hemos extraído del lienzo), y el bucle **`for`** para los valores **`y`** va de 0 a 261 (el número total de píxeles verticales de la imagen que estamos procesando).

Figura 11-19: *Imagen negativa*

Después de que se procesa cada píxel, los datos de la imagen de la variable **info** se insertan en el lienzo con el método **putImageData()**. Esta nueva imagen se ubica en la misma posición que la original, reemplazando la mitad izquierda con el negativo que acabamos de crear.

Origen cruzado

Una aplicación de origen cruzado (*cross-origin* en Inglés) es una aplicación que está localizada en un domino y accede a recursos de otros. Debido a cuestiones de seguridad, algunas API restringen el acceso de origen cruzado. En el caso de la API Canvas, no se puede obtener ninguna información del elemento **<canvas>** después de que se haya dibujado una imagen de otro dominio.

Estas restricciones se pueden evitar con una tecnología llamada *CORS* (*Cross-Origin Resource Sharing*). CORS define un protocolo para servidores con el objetivo de compartir sus recursos con otros orígenes. El acceso de un origen a otro debe ser autorizado por el servidor. La autorización se realiza declarando en el servidor los orígenes (dominios) que tienen permitido acceder a los recursos. Esto se hace en la cabecera enviada por el servidor que aloja el archivo que procesa la solicitud. Por ejemplo, si nuestra aplicación está localizada en el dominio www.domain1.com y accede a recursos en el dominio www.domain2.com, este segundo servidor debe estar configurado para declarar el origen www.domain1.com como un origen válido para la solicitud.

CORS facilita varias cabeceras a incluir como parte de la cabecera HTTP enviada por el servidor, pero la única requerida es **Access-Control-Allow-Origin**. Esta cabecera indica qué orígenes (dominios) pueden acceder a los recursos del servidor. Se puede declarar el carácter * para permitir solicitudes desde cualquier origen.

IMPORTANTE: su servidor debe estar configurado para enviar cabeceras HTTP CORS con cada solicitud para permitir a las aplicaciones acceder a sus archivos. Una manera fácil de lograrlo es agregar una nueva línea al archivo **.htaccess**. La mayoría de los servidores incluyen este archivo de configuración en el directorio raíz de todo sitio web. La sintaxis es **Header set CORS-Header value** (por ejemplo, **Header set Access-Control-Allow-Origin ***). Para obtener más información sobre cómo agregar cabeceras HTTP a su servidor, visite nuestro sitio web y siga los enlaces de este capítulo.

Agregar cabeceras HTTP a la configuración del servidor es solo uno de los pasos que tenemos que seguir para convertir nuestro código en una aplicación de origen cruzado. También tenemos que declarar el recurso como un recurso de origen cruzado usando el atributo **crossOrigin**.

```
var canvas, imagen;
function iniciar() {
  var elemento = document.getElementById("canvas");
  canvas = elemento.getContext("2d");

  imagen = document.createElement("img");
  imagen.crossOrigin = "anonymous";
  imagen.src = "http://www.formasterminds.com/content/nieve.jpg";
```

```
  imagen.addEventListener("load", modimagen);
}
function modimagen() {
  canvas.drawImage(imagen, 0, 0);
  var info = canvas.getImageData(0, 0, 175, 262);
  var pos;
  for (var x = 0; x < 175; x++) {
    for (var y = 0; y < 262; y++) {
      pos = (info.width * 4 * y) + (x * 4);
      info.data[pos] = 255 - info.data[pos];
      info.data[pos+1] = 255 - info.data[pos+1];
      info.data[pos+2] = 255 - info.data[pos+2];
    }
  }
  canvas.putImageData(info, 0, 0);
}
window.addEventListener("load", iniciar);
```

Listado 11-27: *Habilitando el acceso de origen cruzado*

El atributo **crossOrigin** acepta dos valores: **anonymous** y **use-credentials**. El primer valor ignora las credenciales, mientras que el segundo valor requiere que se envíen las credenciales en la solicitud. Las credenciales son compartidas automáticamente por el cliente y el servidor usando cookies. Para poder usar credenciales, tenemos que incluir una segunda cabecera en el servidor llamada **Access-Control-Allow-Credentials** con el valor **true**.

En el Listado 11-27 solo se ha realizado una modificación con respecto al ejemplo anterior: agregamos el atributo **crossOrigin** al elemento **<img>** para declarar la imagen como un recurso de origen cruzado. Ahora podemos ejecutar el código en nuestro ordenador y usar la imagen del servidor sin infringir las políticas de un solo origen (las cabeceras CORS ya se han agregado al servidor www.formasterminds.com).

IMPORTANTE: el atributo **crossOrigin** se tiene que declarar antes que el atributo **src** para configurar la fuente como de origen cruzado. Por supuesto, el atributo también se puede declarar en la etiqueta de apertura de los elementos **<img>**, **<video>** y **<audio>**, como cualquier otro atributo HTML.

Extrayendo datos

El método **getImageData()** estudiado anteriormente devuelve un objeto que se puede procesar a través de sus propiedades (**width**, **height**, y **data**) o utilizar como tal mediante el método **putImageData()**. El propósito de estos métodos es ofrecer acceso al contenido del lienzo y devolver los datos al mismo lienzo o a otro después de que se han procesado. Pero a veces esta información puede ser necesaria para otros propósitos: asignarlos como fuente de un elemento **<img>**, enviarlos al servidor o almacenarlos en una base de datos. La API Canvas incluye los siguientes métodos para obtener el contenido del lienzo en un formato de datos que podemos usar para realizar estas tareas.

toDataURL(tipo**)**—Este método devuelve datos en formato data:url, que contiene una representación del contenido del lienzo en el formato de imagen especificado por el atributo **tipo** y una resolución de 96 dpi. Si no se declara el tipo, los datos se devuelven en formato PNG. Los posibles valores para el atributo son **image/jpeg** e **image/png**.

toBlob(función, tipo**)**—Este método devuelve un objeto con un blob que contiene una representación del contenido del lienzo en el formato especificado por el atributo **tipo** y una resolución de 96 dpi. El primer atributo es la función a cargo de procesar el objeto. Si el tipo no se declara, el blob se devuelve en el formato PNG. Los posibles valores para el atributo son **image/jpeg** e **image/png**.

El siguiente documento agrega una caja al lado del elemento **<canvas>** para mostrar la imagen producida a partir el contenido del lienzo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Canvas</title>
  <style>
    #cajacanvas, #cajadatos {
      float: left;
      width: 400px;
      height: 300px;
      padding: 10px;
      margin: 10px;
      border: 1px solid;
    }
    .recuperar {
      clear: both;
    }
  </style>
  <script src="canvas.js"></script>
</head>
<body>
  <section id="cajacanvas">
    <canvas id="canvas" width="400" height="300"></canvas>
  </section>
  <section id="cajadatos"></section>
  <div class="recuperar"></div>
</body>
</html>
```

Listado 11-28: *Creando un documento para mostrar una imagen con el contenido del lienzo*

El código para este ejemplo dibuja una imagen en el lienzo, extrae el contenido y genera un elemento **<img>** para mostrarlo en la pantalla.

```
function iniciar() {
  var elemento = document.getElementById("canvas");
  var ancho = elemento.width;
  var altura = elemento.height;
  elemento.addEventListener("click", copiarimagen);

  var canvas = elemento.getContext("2d");
  canvas.beginPath();
  canvas.arc(ancho / 2, altura / 2, 150, 0, Math.PI * 2, false);
  canvas.clip();
```

```
  var imagen = document.createElement("img");
  imagen.src = "nieve.jpg";
  imagen.addEventListener("load", function() {
    canvas.drawImage(imagen, 0, 0, ancho, altura);
  });
}
function copiarimagen() {
  var elemento = document.getElementById("canvas");
  var datos = elemento.toDataURL();

  var cajadatos = document.getElementById("cajadatos");
  cajadatos.innerHTML = '<img src="' + datos + '">';
}
window.addEventListener("load", iniciar);
```

Listado 11-29: *Creando una imagen con el contenido del lienzo*

El código del Listado 11-29 crea un área de recorte circular y luego dibuja una imagen en el lienzo. El efecto genera una imagen circular, como la de un retrato. Cuando el usuario hace clic en el lienzo, extraemos esta imagen del lienzo con el método **toDataURL()** y usamos los datos que devuelve el método como la fuente de un nuevo elemento **<img>**. El elemento procesa los datos y muestra la imagen en la pantalla.

Figura 11-20: *Imagen creada desde el contenido del lienzo*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 11-28 y un archivo JavaScript llamado canvas.js con el código del Listado 11-29. Suba los archivos, incluido el archivo nieve.jpg, a su servidor o servidor local y abra el documento en su navegador. Haga clic en el lienzo para extraer el contenido y mostrar la imagen en la caja de la derecha.

Lo básico: Data:url y blobs son dos formatos de distribución de datos diferentes. El formato data:url se denomina oficialmente *Data URI Scheme*. Este formato es simplemente una cadena de texto que representa los datos necesarios para recrear el recurso y, por lo tanto, se puede usar como fuente de elementos HTML, como lo demuestra el ejemplo del Listado 11-29. Los blobs, por otro lado, son bloques que contienen datos sin procesar. Estudiaremos los blobs en el Capítulo 16.

11.4 Animaciones

No existe ningún método que nos ayuda a animar figuras en el lienzo, y no hay ningún procedimiento predeterminado para hacerlo. Simplemente tenemos que borrar el área del lienzo que queremos animar, dibujar las figuras en esa área, y repetir el proceso una y otra vez. Una vez que se han dibujado las figuras, no se pueden mover y solo podemos construir una animación borrando el área y dibujando las figuras nuevamente en una posición diferente. Por esta razón, en juegos o aplicaciones que contienen una cantidad importante de figuras a animar, es mejor usar imágenes en lugar de figuras construidas con trazados complejos (por ejemplo, los juegos usan imágenes PNG).

Animaciones simples

Existen varias técnicas para crear animaciones: algunas son simples y otras más complejas, dependiendo de la aplicación. En el siguiente ejemplo vamos a implementar una técnica de animación básica usando el método **clearRect()** introducido anteriormente para borrar el lienzo y dibujar nuevamente las figuras (el código asume que estamos usando el documento del Listado 11-1 con un elemento **<canvas>** de 500 píxeles por 300 píxeles).

```
var canvas;
function iniciar() {
  var elemento = document.getElementById("canvas");
  canvas = elemento.getContext("2d");
  window.addEventListener("mousemove", animacion);
}
function animacion(evento) {
  canvas.clearRect(0, 0, 500, 300);

  var xraton = evento.clientX;
  var yraton = evento.clientY;
  var xcentro = 220;
  var ycentro = 150;
  var ang = Math.atan2(xraton - xcentro, yraton - ycentro);
  var x = xcentro + Math.round(Math.sin(ang) * 10);
  var y = ycentro + Math.round(Math.cos(ang) * 10);

  canvas.beginPath();
  canvas.arc(xcentro, ycentro, 20, 0, Math.PI * 2, false);
  canvas.moveTo(xcentro + 70, 150);
  canvas.arc(xcentro + 50, 150, 20, 0, Math.PI * 2, false);
  canvas.stroke();

  canvas.beginPath();
  canvas.moveTo(x + 10, y);
  canvas.arc(x, y, 10, 0, Math.PI * 2, false);

  canvas.moveTo(x + 60, y);
  canvas.arc(x + 50, y, 10, 0, Math.PI * 2, false);
  canvas.fill();
}
window.addEventListener("load", iniciar);
```

Listado 11-30: *Creando una animación*

Esta animación consta de un par de ojos que miran continuamente al puntero del ratón. Para mover los ojos tenemos que actualizar sus posiciones cada vez que se desplaza el ratón. Esto lo logramos detectando la posición del ratón en la ventana con el evento **mousemove**. Cada vez que se desencadena el evento, se llama a la función **animacion()**. Esta función limpia el lienzo con la instrucción **clearRect(0, 0, 500, 300)** y luego inicializa las variables. La posición del ratón se captura mediante las propiedades **clientX** y **clientY** y las coordenadas del primer ojo se almacenan en las variables **xcentro** y **ycentro**.

Después de obtener los valores iniciales, es hora de calcular el resto de los valores. Usando la posición del ratón y el centro del ojo izquierdo, calculamos el ángulo de la línea invisible que va desde un punto a otro usando el método **atan2()** del objeto **Math** (ver Capítulo 6). Este ángulo se usa para calcular el punto al centro del iris con la fórmula **xcentro + Math.round(Math.sin(ang) × 10)**. El número **10** en la fórmula representa la distancia desde el centro del ojo al centro del iris (el iris no se ubica en el centro del ojo, sino cerca del borde).

Con estos valores, podemos comenzar a dibujar los ojos en el lienzo. El primer trazado son dos círculos que representa los ojos. El método **arc()** para el ojo izquierdo se posiciona con los valores de **xcentro** e **ycentro**, y el círculo para el ojo derecho se genera 50 píxeles hacia la derecha usando la instrucción **arc(xcentro + 50, 150, 20, 0, Math.PI * 2, false)**.

La animación se crea con el segundo trazado. Este trazado usa las variables **x** e **y** con la posición calculada anteriormente a partir del ángulo. Ambos iris se dibujan como círculos sólidos de color negro mediante el método **fill()**.

El proceso se repite y los valores se vuelven a calcular cada vez que se desencadena el evento **mousemove** (cada vez que el usuario mueve el ratón).

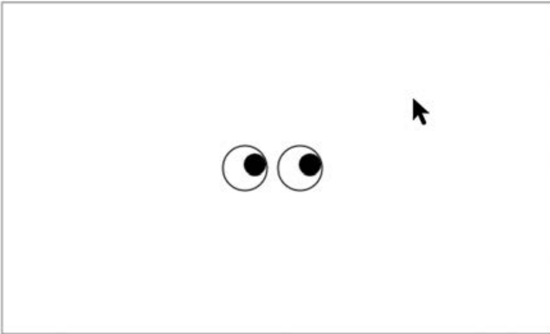

Figura 11-21: *Animación simple*

Hágalo usted mismo: copie el código del Listado 11-30 dentro del archivo JavaScript llamado canvas.js y abra el documento del Listado 11-1 en su navegador. Mueva el ratón alrededor de los círculos. Debería ver los ojos moverse siguiendo el puntero.

IMPORTANTE: en este ejemplo, la distancia se ha calculado sin tener en cuenta la posición del lienzo en la pantalla. Esto se debe a que el elemento **<canvas>** del documento del Listado 11-1 se ha creado en la esquina superior izquierda de la página y, por lo tanto, el origen del lienzo es el mismo que el origen del documento. Para obtener más información sobre las propiedades **clientX** y **clientY**, vea el Capítulo 6.

Animaciones profesionales

El bucle creado para la animación del ejemplo anterior se ha generado mediante el evento **mousemove**. En una animación profesional, los bucles se controlan desde el código y funcionan todo el tiempo, independientemente de la actividad del usuario. En el Capítulo 6, hemos estudiado dos métodos de JavaScript para crear un bucle: **setInterval()** y **setTimeout()**. Estos métodos ejecutan una función después de un cierto período de tiempo. Trabajando con estos métodos, podemos producir animaciones sencillas, pero estas animaciones no estarán sincronizadas con el navegador, causando retrasos que no se toleran en aplicaciones profesionales. Con el propósito de resolver este problema, los navegadores incluyen una pequeña API con solo dos métodos, uno para generar un nuevo ciclo de un bucle y el otro para cancelarlo.

> **requestAnimationFrame(**función**)**—Este método indica al navegador que se debería ejecutar la función entre paréntesis. El navegador llama a la función cuando está listo para actualizar la ventana, sincronizando la animación con la ventana del navegador y la pantalla del ordenador. Podemos asignar este método a una variable y luego cancelar el proceso llamando al método **cancelAnimationFrame()** con el nombre de la variable entre paréntesis.

El método **requestAnimationFrame()** trabaja como el método **setTimeout()**; tenemos que volver a llamarlo en cada ciclo del bucle. La implementación es sencilla, pero el código de una animación profesional requiere cierta organización que solo puede lograrse implementando patrones de programación avanzados. Para nuestro ejemplo, vamos a usar un objeto global y distribuir las tareas entre varios métodos. El siguiente es el documento con el elemento **<canvas>** requerido para presentar los dibujos en la pantalla.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Animaciones</title>
  <style>
    #cajacanvas {
      width: 600px;
      margin: 100px auto;
    }
    #canvas {
      border: 1px solid #999999;
    }
  </style>
  <script src="canvas.js"></script>
</head>
<body>
  <div id="cajacanvas">
    <canvas id="canvas" width="600" height="400"></canvas>
  </div>
</body>
</html>
```

Listado 11-31: *Creando el documento para mostrar una animación profesional*

Los estilos CSS del documento del Listado 11-31 tienen el propósito de centrar el lienzo en la pantalla y facilitar un borde para identificar sus límites. El documento también carga un archivo JavaScript para el siguiente código.

```
var mijuego = {
  canvas: {
    ctx: "",
    desplazamientox: 0,
    desplazamientoy: 0
  },
  nave: {
    x: 300,
    y: 200,
    moverx: 0,
    movery: 0,
    velocidad: 1
  },
  iniciar: function() {
    var elemento = document.getElementById("canvas");
    mijuego.canvas.ctx = elemento.getContext("2d");
    mijuego.canvas.desplazamientox = elemento.offsetLeft;
    mijuego.canvas.desplazamientoy = elemento.offsetTop;
    document.addEventListener("click", function(evento) {
      mijuego.control(evento);
    });
    mijuego.bucle();
  },
  bucle: function() {
    if (mijuego.nave.velocidad) {
      mijuego.procesar();
      mijuego.detectar();
      mijuego.dibujar();
      requestAnimationFrame(function() {
        mijuego.bucle();
      });
    } else {
      mijuego.canvas.ctx.font = "bold 36px verdana, sans-serif";
      mijuego.canvas.ctx.fillText("GAME OVER", 182, 210);
    }
  },
  control: function(evento) {
    var distanciax = evento.clientX - (mijuego.canvas.desplazamientox +
mijuego.nave.x);
    var distanciay = evento.clientY - (mijuego.canvas.desplazamientoy +
mijuego.nave.y);
    var ang = Math.atan2(distanciax, distanciay);
    mijuego.nave.moverx = Math.sin(ang);
    mijuego.nave.movery = Math.cos(ang);
    mijuego.nave.velocidad += 1;
  },
  dibujar: function() {
    mijuego.canvas.ctx.clearRect(0, 0, 600, 400);
    mijuego.canvas.ctx.beginPath();
    mijuego.canvas.ctx.arc(mijuego.nave.x, mijuego.nave.y, 20, 0,
Math.PI / 180 * 360, false);
```

```
    mijuego.canvas.ctx.fill();
  },
  procesar: function() {
    mijuego.nave.x += mijuego.nave.moverx * mijuego.nave.velocidad;
    mijuego.nave.y += mijuego.nave.movery * mijuego.nave.velocidad;
  },
  detectar: function() {
    if (mijuego.nave.x < 0 || mijuego.nave.x > 600 || mijuego.nave.y <
0 || mijuego.nave.y > 400) {
      mijuego.nave.velocidad = 0;
    }
  }
};
window.addEventListener("load", function() {
  mijuego.iniciar();
});
```

Listado 11-32: *Creando un videojuego en 2D*

Una animación profesional debería evitar variables y funciones globales, y concentrar el código dentro de un único objeto global. En el ejemplo del Listado 11-32, hemos llamado a este objeto **mijuego**. Todas las propiedades y métodos necesarios para crear un pequeño videojuego se declaran dentro de este objeto.

Nuestro juego incluye una nave espacial negra que se mueve en la dirección que indica el clic del ratón. El objetivo es cambiar la dirección de la nave para que colisione con los muros. Cada vez que se modifica la dirección, la velocidad de la nave se incrementa, lo que hace que sea cada vez más difícil mantenerla dentro del lienzo.

La organización necesaria para esta clase de aplicación siempre incluye ciertos elementos esenciales. Tenemos que declarar los valores iniciales, controlar el bucle de la animación y distribuir el resto de las tareas en varios métodos. En el ejemplo del Listado 11-32, esta organización se consigue con la inclusión de un total de seis métodos: **iniciar()**, **bucle()**, **control()**, **dibujar()**, **procesar()**, y **detectar()**.

Comenzamos declarando las propiedades **canvas** y **nave**. Estas propiedades contienen objetos con información esencial para el juego. El objeto **canvas** tiene tres propiedades: **ctx** para almacenar el contacto del lienzo, y **desplazamientox** y **desplazamientoy** para almacenar la posición del elemento **<canvas>** relacionada con la página. El objeto **nave**, por otro lado, tiene cinco propiedades: **x** e **y** para almacenar las coordenadas de la nave, **moverx** y **movery** para determinar la dirección, y **velocidad** para almacenar la velocidad actual de la nave. Estas son propiedades importantes necesarias en casi todas las demás secciones del código, pero algunos de sus valores todavía se tienen que inicializar. Como hemos hecho en ejemplos anteriores, este trabajo lo realiza el método **iniciar()**. Este método se llama mediante el evento **load** cuando se carga el documento y es responsable de asignar todos los valores necesarios para iniciar la aplicación.

La primera tarea del método **iniciar()** es obtener una referencia para el contexto del lienzo y la posición del elemento en la ventana usando las propiedades **offsetLeft** y **offsetTop**. Luego, se agrega un listener al evento **click** para responder cuando el usuario hace clic en cualquier parte del documento.

El método **loop()** es el segundo más importante en el alineamiento de la organización de una aplicación profesional. Este método se llama a sí mismo una y otra vez mientras se ejecuta la aplicación, creando un bucle que pasa por cada parte del proceso. En nuestro ejemplo, este

proceso se divide en tres métodos: **procesar()**, **detectar()** y **dibujar()**. El método **procesar()** calcula la nueva posición de la nave de acuerdo con la dirección actual y la velocidad, el método **detectar()** compara las coordenadas de la nave con los límites del lienzo para determinar si la nave ha chocado contra los muros, y el método **dibujar()** dibuja la nave en el lienzo. El bucle ejecuta estos métodos uno por uno y luego se llama a sí mismo con el método **requestAnimationFrame()**, tras lo que comienza un nuevo ciclo.

Lo único que nos queda por hacer para finalizar la estructura básica de nuestra aplicación es controlar las respuestas del usuario. En nuestro juego, esto se realiza con el método **control()**. Cuando el usuario hace clic en cualquier parte del documento, se llama a este método para calcular la dirección de la nave. La fórmula es similar a la utilizada en ejemplos anteriores. Primero, calculamos la distancia desde la nave al lugar donde ha ocurrido el evento, luego obtenemos el ángulo de la línea invisible entre estos dos puntos y finalmente se obtiene la dirección en coordenadas mediante los métodos **sin()** y **cos()**, que se almacena en las propiedades **moverx** y **movery**. La última instrucción del método **control()** incrementa la velocidad de la nave para hacer nuestro juego más interesante.

El juego comienza en cuanto se carga el documento y finaliza cuando la nave choca contra un muro. Para hacer que esta aplicación parezca más un videojuego, agregamos una instrucción **if** en el bucle que controla la condición de la nave. Esta condición la determina el valor de la velocidad. Cuando el método **detectar()** determina que las coordenadas de la nave están fuera de los límites del lienzo, la velocidad se reduce a **0**. Este valor vuelve falsa la condición del bucle y se muestra el mensaje "GAME OVER" en la pantalla.

Figura 11-22: *Videojuego sencillo*

11.5 Vídeo

Al igual que las animaciones, no existe un método particular para mostrar vídeo en un elemento **<canvas>**. La única manera de hacerlo es tomar cada cuadro del vídeo desde el elemento **<video>** y dibujarlo como una imagen en el lienzo usando el método **drawImage()**. El siguiente documento incluye un pequeño código JavaScript que convierte el lienzo en un espejo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Video en el Lienzo</title>
```

```
  <style>
    section {
      float: left;
    }
  </style>
  <script>
    var canvas, video;
    function iniciar() {
      var elemento = document.getElementById("canvas");
      canvas = elemento.getContext("2d");
      video = document.getElementById("medio");

      canvas.translate(483, 0);
      canvas.scale(-1, 1);
      setInterval(procesarcuadros, 33);
    }
    function procesarcuadros() {
      canvas.drawImage(video, 0, 0);
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <video id="medio" width="483" height="272" autoplay>
      <source src="trailer.mp4">
      <source src="trailer.ogg">
    </video>
  </section>
  <section>
    <canvas id="canvas" width="483" height="272"></canvas>
  </section>
</body>
</html>
```

Listado 11-33: *Mostrando un vídeo en el lienzo*

Como hemos explicado antes, el método **drawImage()** puede recibir tres tipos de fuentes: una imagen, un vídeo, u otro lienzo. Por esta razón, para mostrar un vídeo en el lienzo solo tenemos que especificar la referencia al elemento **<video>** como el primer atributo del método. Sin embargo, debemos considerar que los vídeos están compuestos de múltiples cuadros y el método **drawImage()** solo es capaz de dibujar un cuadro a la vez. Por esta razón, para mostrar el vídeo completo en el lienzo, tenemos que repetir el proceso para cada cuadro. En el código del Listado 11-33, el método **setInterval()** se usa para llamar a la función **procesarcuadros()** cada 33 milisegundos. Esta función ejecuta el método **drawImage()** con el vídeo como fuente (el tiempo declarado para **setInterval()** es el tiempo aproximado que tarda cada cuadro en mostrarse en pantalla).

Una vez que la imagen del cuadro se dibuja en el lienzo, está asociada a sus propiedades y, por lo tanto, se puede procesar como cualquier otro contenido. En nuestro ejemplo, la imagen se invierte para crear un efecto espejo. El efecto se crea con la aplicación del método **scale()** en el contexto (todo lo dibujado en el lienzo se invierte).

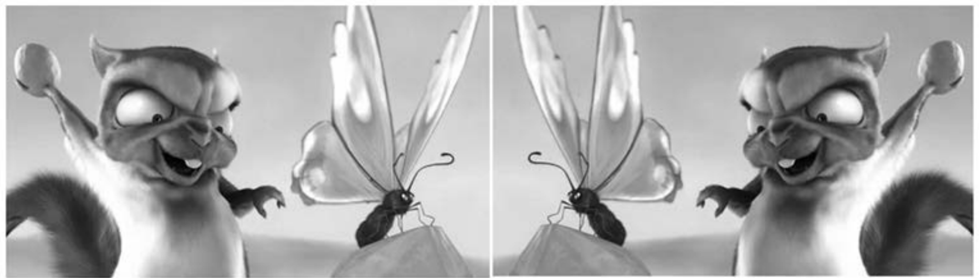

Figura 11-23: *Imagen espejo de un vídeo*
© Derechos Reservados 2008, Blender Foundation / www.bigbuckbunny.org

Aplicación de la vida real

Existen millones de cosas que podemos hacer con un lienzo, pero siempre es fascinante ver lo lejos que podemos llegar combinando métodos de diferentes API. El siguiente ejemplo describe cómo hacer una foto con la cámara y mostrarla en la pantalla usando un elemento **`<canvas>`**.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Aplicación Instantáneas</title>
  <style>
    section {
      float: left;
      width: 320px;
      height: 240px;
      border: 1px solid #000000;
    }
  </style>
  <script>
    var video, canvas;
    function iniciar() {
      var promesa = navigator.mediaDevices.getUserMedia({video: true});
      promesa.then(exito);
      promesa.catch(mostrarerror);
    }
    function exito(transmision) {
      var elemento = document.getElementById("canvas");
      canvas = elemento.getContext("2d");
      var reproductor = document.getElementById("reproductor");
      reproductor.addEventListener("click", instantanea);

      video = document.getElementById("medio");
      video.srcObject = transmision;
      video.play();
    }
    function mostrarerror(evento) {
      console.log("Error: " + evento.name);
    }
    function instantanea() {
```

```
      canvas.drawImage(video, 0, 0, 320, 240);
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section id="reproductor">
    <video id="medio" width="320" height="240"></video>
  </section>
  <section>
    <canvas id="canvas" width="320" height="240"></canvas>
  </section>
</body>
</html>
```

Listado 11-34: *Programando una aplicación para tomar una foto*

Este documento incluye algunos estilos CSS, el código JavaScript y algunos elementos HTML, incluidos los elementos **<video>** y **<canvas>**. El elemento **<canvas>** se declara como lo hacemos usualmente, pero no especificamos la fuente del elemento **<video>** porque vamos a usarlo para mostrar el vídeo de la cámara.

El código es tan sencillo como efectivo. Nuestra función estándar **iniciar()** llama al método **getUserMedia()** tan pronto como el documento se carga para acceder a la cámara. En caso de éxito, se llama a la función **exito()**. La mayoría del trabajo lo realiza esta función. La función obtiene el contexto del lienzo, agrega un listener para el evento **click** a la caja del vídeo y asigna el vídeo de la cámara al elemento **<video>**. Al final, el vídeo se reproduce con el método **play()**.

En este momento, el vídeo de la cámara se está mostrando en la caja de la izquierda de la pantalla. Para hacer una foto y presentarla en la caja de la derecha, hemos creado la función **instantanea()**. Esta función responde al evento **click**. Cuando el usuario hace clic en el vídeo, la función ejecuta el método **drawImage()** con una referencia al vídeo y los valores correspondientes al tamaño del elemento **<canvas>**. El método toma el cuadro actual y lo dibuja en el lienzo para mostrar la instantánea en la pantalla.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 11-34. Abra el documento en su navegador y autorice al navegador a acceder a la cámara. Haga clic en el vídeo. La imagen se debería dibujar en el lienzo. Recuerde subir el archivo al servidor antes de probarlo.

Capítulo 12

WebGL

12.1 Lienzo en 3D

WebGL es una API de bajo nivel que trabaja con el elemento **<canvas>** para crear gráficos en 3D para la Web. Utiliza la GPU (Unidad de Procesamiento Gráfico) de la tarjeta de vídeo para realizar algunas operaciones y liberar a la CPU (Unidad Central de Procesamiento) del trabajo duro. Esta API está basada en OpenGL, una librería reconocida que fue desarrollada por Silicon Graphics Inc. y se ha aplicado en la creación de videojuegos en 3D y aplicaciones desde 1992. Estas características convierten a WebGL en una API muy fiable y eficaz, y esta es probablemente la razón por la que se ha vuelto la API 3D estándar para la Web.

WebGL se ha implementado en casi todos los navegadores compatibles con HTML5, pero su complejidad ha forzado a los desarrolladores a trabajar con librerías JavaScript creadas sobre la misma en lugar de hacerlo directamente con la API. Esta complejidad va más allá del hecho que WebGL no incorpora métodos nativos para realizar operaciones 3D básicas, más bien está relacionada con su naturaleza de bajo nivel, lo cual requiere la utilización de códigos externos programados en un lenguaje llamado *GLSL* (*OpenGL Shading Language*), que facilitan herramientas esenciales para la producción de las imágenes en pantalla. Por estas razones, desarrollar una aplicación con WebGL siempre requiere el uso de librerías externas como glMatrix para calcular matrices y vectores (github.com/toji/gl-matrix), o librerías más generales como Three.js (www.threejs.org), GLGE (www.glge.org), SceneJS (www.scenejs.org), entre otras.

IMPORTANTE: debido a las características de WebGL, y también a la extensión de la materia, no estudiaremos cómo aplicar WebGL. En cambio, en este capítulo aprenderá a generar y trabajar con gráficos en 3D usando una librería externa sencilla llamada *Three.js*.

12.2 Three.js

Three.js probablemente sea la librería más popular y completa para la generación de gráficos en 3D usando WebGL. Esta librería trabaja sobre WebGL, simplificando la mayoría de las tareas y ofreciendo todas las herramientas que necesitamos para controlar cámaras, luces, objetos, texturas, y crear un mundo en 3D. Es fácil de instalar; solo tenemos que descargar el paquete de archivos desde www.threejs.org e incluir el archivo three.min.js en nuestros documentos.

IMPORTANTE: este capítulo no intenta ser un manual de Three.js. Para obtener una referencia completa, visite el sitio web de la librería y lea la documentación.

Hágalo usted mismo: visite www.threejs.org y haga clic en Download para descargar el paquete con la librería. Este paquete incluye múltiples archivos y ejemplos, pero solo necesita el archivo three.min.js que se encuentra dentro del directorio build. Copie este archivo dentro del directorio de su proyecto para poder incorporarlo a sus documentos.

Renderer

El renderer es la superficie en la que se dibujan los gráficos. Three.js utiliza un elemento **<canvas>** con un contexto WebGL para generar escenas 3D en el navegador (se encarga de crear el contexto con el parámetro **webgl** por nosotros). La librería incluye el siguiente constructor para obtener y configurar este renderer.

WebGLRenderer(parámetros**)**—Este constructor devuelve un objeto **WebGLRenderer** con las propiedades y los métodos necesarios para configurar la superficie de dibujo y generar los gráficos en la pantalla. El atributo **parámetros** tiene que ser especificado como un objeto. Las propiedades disponibles para este objeto son **canvas** (el elemento **<canvas>**), **precision** (los valores **highp**, **mediump**, **lowp**), **alpha** (un valor booleano), **premultipliedAlpha** (un valor booleano), **antialias** (un valor booleano), **stencil** (un valor booleano), **preserveDrawingBuffer** (un valor booleano), **clearColor** (un valor entero), y **clearAlpha** (un valor decimal).

El objeto **WebGLRenderer** incluye varias propiedades y métodos para configurar el renderer. Los siguientes son los más usados.

setSize(ancho, altura**)**—Este método redimensiona el lienzo a los valores de los atributos **ancho** y **altura**.

setViewport(x, y, ancho, altura**)**—Este método determina el área del lienzo que se usará para crear el gráfico de la escena. El tamaño del área usado por WebGL no tiene por qué ser el mismo que la superficie de dibujo. Los atributos **x** e **y** indican las coordenadas en las que comienza el área de visualización, mientras que **ancho** y **altura** indican su tamaño.

setClearColor(color, alfa**)**—Este método declara el color de la superficie en valores hexadecimales. El atributo **alfa** declara la opacidad (desde **0.0** a **1.0**).

render(escena, cámara, destino, limpiar**)**—Este método crea la gráfica de la escena usando una cámara. Los atributos **escena** y **cámara** son objetos que se presentan en la escena y en la cámara, el atributo **destino** indica dónde se creará la gráfica de la escena (no es necesario para el lienzo declarado por el constructor), y el atributo booleano **limpiar** determina si el lienzo se debe borrar antes o no.

IMPORTANTE: la librería también incluye el constructor **CanvasRenderer()**. Este constructor devuelve un objeto para trabajar con un contexto 2D cuando el navegador no ofrece soporte para WebGL. Este renderer no trabaja con la GPU, por lo que no se recomienda en aplicaciones exigentes.

Escena

Una escena es un objeto global que contiene el resto de los objetos que representan cada elemento del mundo 3D, como la cámara, luces, mallas, etc. Three.js provee un constructor sencillo para generar una escena.

scene()—Este constructor devuelve un objeto que representa la escena. Un objeto **Scene** provee los métodos **add()** y **remove()** para agregar y eliminar elementos en la escena.

La escena establece un espacio tridimensional que nos ayuda a localizar todos los elementos en el mundo virtual. Las coordenadas de este espacio se identifican con las letras **x**, **y** y **z**. Cada elemento tendrá sus propias coordenadas y su posición en el mundo 3D, tal como se representa a continuación.

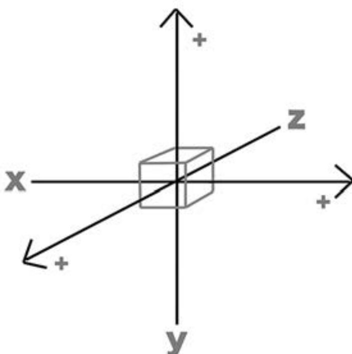

Figura 12-1: *Cubo en un sistema de coordenadas en 3D*

Si incrementamos el valor de la coordenada **x** de un elemento, ese elemento será desplazado a una nueva posición en el eje **x**, pero mantendrá la misma posición en el resto de los ejes, como muestra la Figura 12-2.

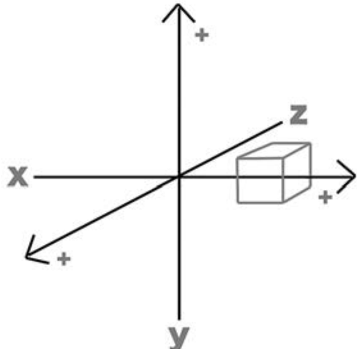

Figura 12-2: *Objeto desplazado en el eje x*

Con cada nuevo elemento del mundo, como la cámara, las luces y los objetos físicos, tenemos que declarar los valores de las tres coordenadas para establecer su posición en la escena. Cuando estas coordenadas no se declaran, el elemento se posiciona en el origen (las coordenadas `0`, `0`, `0`).

Debido a que no contamos con parámetros con los que determinar el tamaño o la escala de un mundo 3D, los valores no tienen unidades de medida: se declaran simplemente como números decimales. La escala se establece mediante la relación entre los elementos ya definidos en el mundo y la cámara.

Cámara

La cámara es una parte esencial de la escena 3D. Determina el punto de vista del espectador (el usuario) y facilita la perspectiva necesaria para que nuestro mundo parezca realista.

Three.js incluye constructores para crear diferentes tipos de cámara. Los siguientes son los más usados.

PerspectiveCamera(fov, aspecto, cerca, lejos**)**—Este constructor devuelve un objeto que representa una cámara con proyección de perspectiva. El atributo **fov** determina el área de visión vertical, **aspecto** declara la proporción, y los atributos **cerca** y **lejos** ponen límites a lo que puede ver la cámara (los puntos más cercanos y lejanos).

OrthographicCamera(izquierda, derecha, superior, inferior, cerca, lejos**)**—Este constructor devuelve un objeto que representa una cámara con proyección ortográfica. Los atributos **izquierda**, **derecha**, **superior**, e **inferior** declaran el plano de frustum correspondiente. Los atributos **cerca** y **lejos** ponen límites a lo que la cámara puede ver (los puntos más cercano y lejano).

Las proyecciones ortográfica y de perspectiva son simplemente diferentes maneras de proyectar el mundo tridimensional en una superficie de dos dimensiones como el lienzo. La proyección de perspectiva es más realista en cuanto a la imitación del mundo real y es la recomendada para animaciones, pero la proyección ortográfica es útil para la visualización de estructuras y dibujos que requieren más detalles porque ignora algunos efectos producidos por el ojo humano.

El objeto `Camera` que devuelven estos constructores incluye un método con el que declarar el vector al que la cámara está apuntando.

lookAt(vector**)**—Este método orienta la cámara hacia un punto de la escena. El atributo **vector** es un objeto `Vector` que contiene las propiedades para los valores de las tres coordenadas **x**, **y**, y **z** (por ejemplo, `{x: 10, y: 10, z: 20}`). Por defecto, la cámara apunta al origen (las coordenadas 0, 0, 0).

Mallas

En un mundo 3D, los objetos físicos se representan a través de mallas. Una malla es una colección de vértices que define una figura. Cada vértice de la malla es un nodo en el espacio tridimensional. Los nodos se unen por líneas invisibles que generan pequeños planos alrededor de la figura. El grupo de planos obtenidos por la intersección de todos los vértices de la malla constituyen la superficie de la figura.

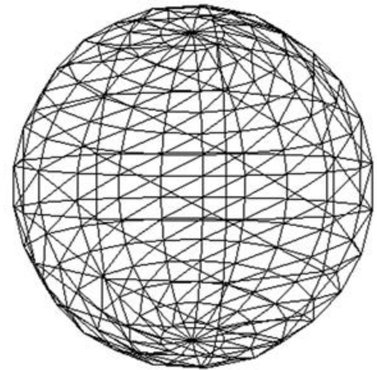

Figura 12-3: *Malla para construir una esfera*

La Figura 12-3 muestra una malla que representa una esfera. Para crear esta malla en particular, tenemos que definir 256 vértices que generan un total de 225 planos. Esto significa que tenemos que declarar las coordenadas **x**, **y**, y **z** 256 veces para crear todos los nodos necesarios para definir una simple esfera. Este ejemplo muestra lo difícil que es definir objetos tridimensionales. Debido a esta complejidad, los diseñadores trabajan con aplicaciones de modelado para generar gráficos en 3D. Programas como Blender (www.blender.org), por ejemplo, exportan mallas completas en formatos especiales que otras aplicaciones pueden leer e implementar en el mundo 3D sin tener que crear los vértices uno por uno (volveremos a hablar de este tema más adelante).

Las mallas, como cualquier otro elemento, se tienen que encapsular en un objeto antes de introducirlas en la escena. La librería ofrece un método que tiene este propósito específico.

Mesh(geometría, material**)**—Este constructor devuelve un objeto que representa una malla. Los atributos **geometría** y **material** son objetos que devuelven los constructores de geometrías y materiales, como veremos a continuación.

Figuras primitivas

Una esfera, como la que se muestra en la Figura 12-3, se llama *primitiva* o *figura primitiva*. Las primitivas son figuras geométricas que tienen una estructura definida. Para algunas aplicaciones, como pequeños videojuegos, las primitivas pueden ser extremadamente útiles porque simplifican el trabajo de diseñadores y desarrolladores. Three.js ofrece varios constructores para crear las figuras primitivas más comunes.

SphereGeometry(radio, segmentos_horizontales, segmentos_verticales, phiStart, phiLength, thetaStart, thetaLength**)**—Este constructor devuelve un objeto que contiene la malla para construir una esfera. El atributo **radio** define el radio de la esfera, los atributos **segmentos_horizontales** y **segmentos_verticales** declaran el número de segmentos incluidos horizontalmente y verticalmente para crear la figura, y la combinación de los atributos **phiStart**, **phiLength**, **thetaStart** y **thetaLength** nos permite generar una esfera incompleta. La mayoría de los atributos son opcionales y tienen valores por defecto.

BoxGeometry(ancho, altura, profundidad, segmentos_horizontales, segmentos_verticales, segmentos_profundidad, materiales, lados**)**—Este constructor devuelve un objeto que contiene una malla para construir un cubo. Los atributos **ancho**, **altura**, y **profundidad** determinan el tamaño de cada lado del cubo, los atributos **segmentos_horizontales**, **segmentos_verticales** y **segmentos_profundidad** establecen el número de segmentos usados para crear los lados del cubo, el atributo **materiales** es un array que contiene diferentes materiales para cada lado, y el atributo **lados** contiene seis valores booleanos para especificar si se generará cada cara o no. La mayoría de los atributos son opcionales y tienen valores por defecto.

CylinderGeometry(radio_superior, radio_inferior, altura, segmentos_horizontales, segmentos_verticales**)**—Este constructor devuelve un objeto que contiene una malla para construir un cilindro. Los atributos **radio_superior** y **radio_inferior** especifican los radios de los extremos superior e inferior del cilindro (diferentes radios construyen un cono), el atributo **altura** declara la altura del cilindro, y los atributos **segmentos_horizontales** y

segmentos_verticales declaran el número de segmentos horizontales y verticales que se usarán para construir la malla.

IcosahedronGeometry(radio, detalles**)**—Este constructor devuelve un objeto que contiene la malla para construir un icosaedro. El atributo **radio** declara el radio y el atributo **detalles** especifica el nivel de detalles. Los niveles de detalle más altos requieren más segmentos. Normalmente, el valor de este atributo oscila entre `0` y no más de `5`, dependiendo del tamaño de la figura y el nivel de detalle que necesitamos.

OctahedronGeometry(radio, detalles**)**—Este constructor devuelve un objeto que contiene la malla para construir un octaedro. El atributo **radio** declara el radio y el atributo **detalles** especifica el nivel de detalles. Los niveles de detalle más altos requieren más segmentos. Normalmente, el valor de este atributo oscila entre `0` y no más de `5`, dependiendo del tamaño de la figura y el nivel de detalle que necesitamos.

TetrahedronGeometry(radio, detalles**)**—Este constructor devuelve un objeto que contiene la malla para construir un tetraedro. El atributo **radio** declara el radio y el atributo **detalles** especifica el nivel de detalles. Los niveles de detalle más altos requieren más segmentos. Normalmente, el valor de este atributo oscila entre `0` y no más de `5`, dependiendo del tamaño de la figura y el nivel de detalle que necesitamos.

PlaneGeometry(ancho, altura, segmentos_horizontales, segmentos_verticales**)**—Este constructor devuelve un objeto que contiene la malla para construir un plano. Los atributos **ancho** y **altura** declaran el ancho y la altura del plano, mientras que los atributos **segmentos_horizontales** y **segmentos_verticales** especifican cuántos segmentos se usarán para construirlo.

CircleGeometry(radio, segmentos, ángulo_inicio, ángulo_final**)**—Este constructor devuelve un objeto que contiene la malla para construir un circulo plano. El atributo **radio** declara el radio del círculo, el atributo **segmentos** especifica el número de segmentos usados para construir la figura, y los atributos **ángulo_inicio** y **ángulo_final** declaran los ángulos en radianes en los que el círculo comienza y finaliza (para obtener un círculo completo, los valores deberían ser `0` y `Math.PI * 2`).

Materiales

WebGL utiliza *shaders* (sombreados) para representar los gráficos 3D en la pantalla. Los *shaders* son códigos programados en el lenguaje GLSL (OpenGL Shading Language) que trabajan directamente con la GPU para producir la imagen en la pantalla. Estos códigos facilitan los niveles correctos de luminosidad y sombras para cada píxel de la imagen, de modo que generan la percepción de un mundo en tres dimensiones. Este concepto complejo Three.js lo oculta detrás de los materiales y las luces. Al definir materiales y luces, Three.js determina cómo se mostrará el mundo 3D en la pantalla usando shaders preprogramados que son de uso común en animaciones en 3D.

Three.js define varios tipos de materiales que se tienen que aplicar de acuerdo a los requerimientos de la aplicación y a los recursos disponibles. Los materiales más realistas requieren mayor poder de procesamiento. El material que elegimos para nuestros proyectos tendrá que compensar el nivel de realismo que nuestra aplicación necesita con la cantidad de procesamiento requerido. Se pueden aplicar diferentes materiales a varios objetos en el mismo mundo. Los siguientes constructores se usan para definirlos.

LineBasicMaterial(parámetros**)**—Este material se usa para crear gráficos de mallas con líneas (por ejemplo, una cuadrícula). El atributo **parámetros** es un objeto que contiene propiedades para la configuración del material. Las propiedades disponibles son `color` (un valor hexadecimal), `opacity` (un valor decimal), `blending` (una constante), `depthTest` (un valor booleano), `linewidth` (un valor decimal), `linecap` (los valores `butt`, `round` o `square`), `linejoin` (los valores `round`, `bevel` o `miter`), `vertexColors` (un valor booleano), y `fog` (un valor booleano).

MeshBasicMaterial(parámetros**)**—Este material grafica la malla con un solo color, sin simular el reflejo de luces. No es un efecto realista, pero puede resultar útil en algunas circunstancias. El atributo **parámetros** es un objeto que contiene las propiedades para la configuración del material. Las propiedades disponibles son `color` (un valor hexadecimal), `opacity` (un valor decimal), `map` (un objeto `Texture`), `lightMap` (un objeto `Texture`), `specularMap` (un objeto `Texture`), `envMap` (un objeto `TextureCube`), `combine` (una constante), `reflectivity` (un valor decimal), `refractionRatio` (un valor decimal), `shading` (una constante), `blending` (una constante), `depthTest` (un valor booleano), `wireframe` (un valor booleano), `wireframelinewidth` (un valor decimal), `vertexColors` (una constante), `skinning` (un valor booleano), `morphTargets` (un valor booleano) y `fog` (un valor booleano).

MeshNormalMaterial(parámetros**)**—Este material define un tono de color para cada plano de la malla. El efecto logrado no es realista porque los planos son discernibles, pero es particularmente útil cuando no se dispone de un ordenador para calcular un material más realista. El atributo **parámetros** es un objeto que contiene las propiedades para la configuración del material. Las propiedades disponibles son `opacity` (un valor decimal), `shading` (una constante), `blending` (una constante), `depthTest` (un valor booleano), `wireframe` (un valor booleano) y `wireframelinewidth` (un valor decimal).

MeshLambertMaterial(parámetros**)**—Este material genera una degradación suave en la superficie de la malla, que produce un efecto de luz reflejándose en el objeto. El efecto es independiente del punto de vista del observador. El atributo **parámetros** es un objeto que contiene propiedades para la configuración del material. Las propiedades disponibles son `color` (un valor hexadecimal), `ambient` (un valor hexadecimal), `emissive` (un valor hexadecimal), `opacity` (un valor decimal), `map` (un objeto `Texture`), `lightMap` (un objeto `Texture`), `specularMap` (un objeto `Texture`), `envMap` (un objeto `TextureCube`), `combine` (una constante), `reflectivity` (un valor decimal), `refractionRatio` (un valor decimal), `shading` (una constante), `blending` (una constante), `depthTest` (un valor booleano), `wireframe` (un valor booleano), `wireframelinewidth` (un valor decimal), `vertexColors` (una constante), `skinning` (un valor booleano), `morphTargets` (un valor booleano), `morphNormals` (un valor booleano) y `fog` (un valor booleano).

MeshPhongMaterial(parámetros**)**—Este material produce un efecto realista con un degradado suave sobre toda la superficie de la malla. El atributo **parámetros** es un objeto que contiene las propiedades para la configuración del material. Las propiedades disponibles son `color` (un valor hexadecimal), `ambient` (un valor hexadecimal), `emissive` (un valor hexadecimal), `specular` (un valor hexadecimal), `shininess` (un valor decimal), `opacity` (un valor decimal), `map` (un objeto `Texture`), `lightMap` (un objeto `Texture`), `bumpMap` (un objeto `Texture`), `bumpScale` (un valor decimal), `normalMap` (un objeto `Texture`), `normalScale` (un objeto `Vector`), `specularMap` (un

objeto **`Texture`**), **`envMap`** (Un objeto **`TextureCube`**), **`combine`** (una constante), **`reflectivity`** (un valor decimal), **`refractionRatio`** (un valor decimal), **`shading`** (una constante), **`blending`** (una constante), **`depthTest`** (un valor booleano), **`wireframe`** (un valor booleano), **`wireframelinewidth`** (un valor decimal), **`vertexColors`** (una constante), **`skinning`** (un valor booleano), **`morphTargets`** (un valor booleano), **`morphNormals`** (un valor booleano) y **`fog`** (un valor booleano).

MeshFaceMaterial()—Este constructor se aplica cuando se han declarado diferentes materiales y texturas para cada lado de la geometría.

ParticleBasicMaterial(parámetros**)**—Este material se designa para graficar partículas (por ejemplo, humo, explosiones, etc.). El atributo **parámetros** es un objeto que contiene las propiedades para la configuración del material. Las propiedades disponibles son **`color`** (un valor hexadecimal), **`opacity`** (un valor decimal), **`map`** (un objeto **`Texture`**), **`size`** (un valor decimal), **`sizeAttenuation`** (un valor booleano) **`blending`** (una constante), **`depthTest`** (un valor booleano), **`vertexColors`** (un valor booleano) y **`fog`** (un valor booleano).

ShaderMaterial(parámetros**)**—Este constructor nos permite incorporar nuestros propios shaders. El atributo **parámetros** es un objeto que contiene propiedades para la configuración del material. Las propiedades disponibles son **`fragmentShader`** (una cadena de caracteres), **`vertexShader`** (una cadena de caracteres), **`uniforms`** (un objeto), **`defines`** (un objeto), **`shading`** (una constante), **`blending`** (una constante), **`depthTest`** (un valor booleano), **`wireframe`** (un valor booleano), **`wireframelinewidth`** (un valor decimal), **`lights`** (un valor booleano), **`vertexColors`** (una constante), **`skinning`** (un valor booleano), **`morphTargets`** (un valor booleano), **`morphNormals`** (un valor booleano) y **`fog`** (un valor booleano).

IMPORTANTE: la mayoría de los parámetros para los constructores de materiales tienen valores por defecto que normalmente son los que requieren diseñadores y desarrolladores. Vamos a demostrar cómo configurar algunos de ellos, pero para obtener una referencia completa, debe leer la documentación oficial de la librería y los ejemplos en www.threejs.org.

Estos constructores comparten propiedades en común que se pueden alterar para modificar aspectos básicos de la configuración.

name—Esta propiedad define el nombre del material. Se define una cadena de texto vacía por defecto.

opacity—Esta propiedad define la opacidad del material. Acepta valores desde `0.0` a `1`. El valor `1` se declara por defecto (completamente opaco).

transparent—Esta propiedad define si el material es transparente o no. El valor por defecto es **`false`**.

visible—Esta propiedad define si el material es invisible o no. El valor por defecto es **`true`**.

side—Esta propiedad define de qué lado de la malla se creará el gráfico. Acepta tres constantes: **`THREE.FrontSide`**, **`THREE.BackSide`** y **`THREE.DoubleSide`**.

Implementación

Toda esta teoría puede ser desalentadora, pero su implementación es muy sencilla. En el siguiente ejemplo se crea una esfera como la de la Figura 12-3. Para dibujarla, vamos a usar un elemento **<canvas>** de 500 píxeles por 400 píxeles, una cámara de perspectiva y un material básico para dibujar el objeto en la pantalla como una esfera de alambre.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Three.js</title>
  <script src="three.min.js"></script>
  <script src="webgl.js"></script>
</head>
<body>
  <section>
    <canvas id="canvas" width="500" height="400"></canvas>
  </section>
</body>
</html>
```

Listado 12-1: *Incluyendo la librería Three.js en el documento*

El paso más importante en cualquier aplicación Three.js es cargar la librería. En el Listado 12-1, el archivo three.min.js se incluye en el documento mediante uno de los elementos **<script>**. El segundo elemento **<script>** se encarga de cargar nuestro propio código JavaScript.

```
function iniciar() {
  var canvas = document.getElementById("canvas");
  var ancho = canvas.width;
  var altura = canvas.height;

  var renderer = new THREE.WebGLRenderer({canvas: canvas});
  renderer.setClearColor(0xFFFFFF);

  var escena = new THREE.Scene();
  var camara = new THREE.PerspectiveCamera(75, ancho / altura, 0.1, 1000);
  camara.position.set(0, 0, 150);

  var geometria = new THREE.SphereGeometry(80, 15, 15);
  var material = new THREE.MeshBasicMaterial({color: 0x000000,
wireframe: true});
  var malla = new THREE.Mesh(geometria, material);
  escena.add(malla);

  renderer.render(escena, camara);
}
window.addEventListener("load", iniciar);
```

Listado 12-2: *Creando una esfera de alambre*

Como siempre, nuestro código incluye la función **iniciar()** para iniciar la aplicación. En esta función, realizamos todo el proceso de creación de un mundo en 3D; configuramos el renderer, la escena, la cámara, y una malla. Primero, el constructor **WebGLRenderer()** crea el renderer. Debido a que todas las propiedades, métodos y constructores que ofrece Three.js son parte del objeto **THREE**, debemos llamar al constructor **WebGLRenderer()** desde este objeto, como muestra el Listado 12-2 (**THREE.WebGLRenderer()**). El constructor recibe la propiedad **canvas** para declarar el elemento **<canvas>** en el documento como la superficie de graficado y devuelve un objeto para representar el renderer. A continuación, declaramos un color de fondo blanco con el método **setClearColor()** y creamos la escena usando el constructor **Scene()**. Este constructor no requiere ningún parámetro. El objeto **Scene** que devuelve se almacena en la variable **escena** para su uso posterior.

Una vez que tenemos la escena, el paso siguiente es crear la cámara. Usando el constructor **PerspectiveCamera()**, obtenemos una cámara con proyección de perspectiva (la recomendada para animaciones). El primer atributo declara un ancho de punto de vista de **75**. Este valor es apropiado para nuestra escena, pero se puede cambiar de acuerdo con la escala del mundo que estemos creando. El segundo parámetro declara la proporción de la cámara igual a la de la superficie de graficado. Este valor se obtiene dividiendo el ancho del lienzo por su altura con las variable **ancho** y **altura** definidas al comienzo del código. Finalmente, el límite más cercano se define como **0.1** y el límite más lejano como **1000**. Estos valores también dependen de la escala que estemos usando en nuestro mundo. En este caso, nuestros objetos no serán mayores de **100** o **150** unidades, por lo que un límite de **1000** es más que suficiente para esta escena pequeña. Los objetos que van más allá de estos límites no se van a dibujar en la pantalla.

La cámara, como cualquier otro elemento del mundo, se localiza inicialmente en el origen (las coordenadas **0**, **0**, **0**). Para poder visualizar la malla creada a continuación, la cámara tiene que moverse a una nueva posición. Esto se realiza con la propiedad **position** y el método **set()**. En nuestro ejemplo, este método establece las coordenadas de la cámara como **0** para **x**, **0** para **y** y **150** para **z**, desplazando efectivamente la cámara 150 unidades en el eje **z** (aprenderemos más acerca de este método a continuación).

Lo último que necesitamos agregar a nuestra escena es la malla. La malla y el material correspondiente se definen primero con los constructores **SphereGeometry()** y **MeshBasicMaterial()**, y luego se usan como los atributos del constructor **Mesh()** para obtener el objeto final. El material se define con la propiedad **wireframe** declarada con el valor **true** para obtener una esfera de alambre en la pantalla en lugar de un objeto sólido. La malla finalmente se agrega a la escena mediante el método **add()** y la escena se grafica en el lienzo usando el método **render()** del objeto **renderer**.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 12-1, y un archivo JavaScript llamado webgl.js con el código del Listado 12-2. Recuerde copiar el archivo three.min.js dentro del directorio de su proyecto. Abra el documento en su navegador. El resultado debería ser similar a la Figura 12-3.

IMPORTANTE: los colores en esta librería se representan mediante números hexadecimales con una sintaxis que requiere agregar el prefijo **0x** a los valores (por ejemplo, **0xFF00FF**).

Transformaciones

Cada elemento del mundo 3D, incluidas la cámara y las luces, se colocan por defecto en el origen de la escena (las coordenadas `0, 0, 0`), y sus tamaños y orientación se determinan mediante las coordenadas de cada uno de sus vértices. Las modificaciones para la cámara y las luces se pueden introducir fácilmente, pero las mallas requieren que recalculemos cada vértice. Para llevar a cabo esta tarea, Three.js ofrece un grupo de propiedades y métodos que pueden realizar las transformaciones más comunes. Usando una pocas propiedades, podemos transferir, rotar y escalar un elemento del mundo sin tener que recalcular miles de vértices.

position—Esta propiedad devuelve un objeto que contiene un vértice que podemos usar para obtener o declarar la posición de un elemento en el mundo 3D. El objeto incluye las propiedades **x**, **y** y **z** para leer o modificar cada coordenada independientemente.

rotation—Esta propiedad devuelve un objeto que contiene un vértice que podemos usar para obtener o declarar el ángulo del elemento en radianes. El objeto incluye las propiedades **x**, **y** y **z** para leer o modificar el ángulo de cada coordenada independientemente.

scale—Esta propiedad devuelve un objeto que contiene un vértice que podemos usar para obtener o declarar la escala del elemento. El objeto incluye las propiedades **x**, **y** y **z** para obtener o modificar la escala de cada coordenada independientemente. Por defecto, se declaran con el valor `1`.

Generalmente, se deben modificar de forma simultánea los valores de las tres coordenadas. Three.js ofrece el método **set()** para simplificar este proceso. El método se puede aplicar a cada propiedad de transformación y los valores se declaran separados por comas, tal como muestra el código del Listado 12-2 (**camara.position.set(0, 0, 150)**). En el siguiente ejemplo, demostramos cómo realizar una rotación.

```
var renderer, escena, camara, malla;
function iniciar() {
  var canvas = document.getElementById("canvas");
  var ancho = canvas.width;
  var altura = canvas.height;
  renderer = new THREE.WebGLRenderer({canvas: canvas});
  renderer.setClearColor(0xFFFFFF);
  escena = new THREE.Scene();
  camara = new THREE.PerspectiveCamera(45, ancho / altura, 0.1, 1000);
  camara.position.set(0, 0, 150);
  var geometria = new THREE.BoxGeometry(50, 50, 50);
  var material = new THREE.MeshBasicMaterial({color: 0x000000,
wireframe: true});
  malla = new THREE.Mesh(geometria, material);
  escena.add(malla);
  renderer.render(escena, camara);
  canvas.addEventListener("mousemove", mover);
}
function mover(evento) {
  malla.rotation.x = evento.pageY * 0.01;
  malla.rotation.z = -evento.pageX * 0.01;
  renderer.render(escena, camara);
}
```

```
window.addEventListener("load", iniciar);
```

Listado 12-3: *Rotando un cubo con el ratón*

El ejemplo del Listado 12-3 muestra cómo transformar una malla dinámicamente. Para una mejor visualización, generamos un cubo con el constructor **`BoxGeometry()`**. El resto del código es similar al del ejemplo anterior, pero ahora se agrega un listener para el evento **`mousemove`** al final de la función **`iniciar()`** para crear una animación sencilla. Cada vez que el ratón se mueve sobre el elemento **`<canvas>`**, se llama a la función **`mover()`** y la malla se rota en los ejes **x** y **z** según la posición del ratón.

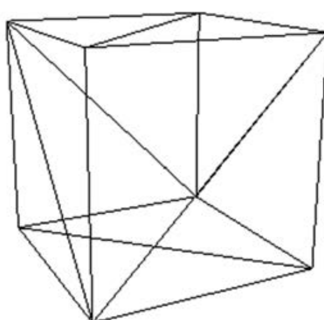

Figura 12-4: *Cubo de alambre animado*

Hágalo usted mismo: copie el código JavaScript del Listado 12-3 dentro del archivo webgl.js creado para el ejemplo anterior y abra el documento del Listado 12-1 en su navegador. Mueva el ratón para rotar el cubo.

Luces

Para crear mallas con materiales que puedan emular la reflexión de la luz, necesitamos generar luces. Three.js ofrece los siguientes constructores para generar una luz.

AmbientLight(color**)**—Esta es una luz global que no se atribuye a ninguna fuente de luz. Se genera con una luz general que refleja cada objeto en la escena. Afecta a todos los objetos del mismo modo. El atributo **color** es el color de la luz en un valor hexadecimal.

DirectionalLight(color, intensidad, distancia**)**—Este tipo de luz se localiza lejos del mundo 3D y afecta a los objetos desde una sola dirección. Los atributos disponibles son **color** (un valor hexadecimal), **intensidad** (un valor decimal) y **distancia** (un valor decimal).

PointLight(color, intensidad, distancia**)**—Este constructor crea una fuente de luz en una ubicación especifica del mundo. Afecta a los objetos en todas direcciones. Los atributos disponibles son **color** (un valor hexadecimal), **intensidad** (un valor decimal) y **distancia** (un valor decimal).

SpotLight(color, intensidad, distancia, sombra**)**—Este constructor crea una fuente de luz direccional en una ubicación específica. Los atributos disponibles son **color** (un valor hexadecimal), **intensidad** (un valor decimal), **distancia** (un valor decimal) y **sombra** (un valor booleano).

El siguiente ejemplo agrega luz a nuestra escena para iluminar el cubo.

```
function iniciar() {
  var canvas = document.getElementById("canvas");
  var ancho = canvas.width;
  var altura = canvas.height;
  var renderer = new THREE.WebGLRenderer({canvas: canvas, antialias: true});
  renderer.setClearColor(0xFFFFFF);
  var escena = new THREE.Scene();
  var camara = new THREE.PerspectiveCamera(45, ancho / altura, 0.1, 1000);
  camara.position.set(0, 0, 150);
  var geometria = new THREE.BoxGeometry(50, 50, 50);
  var material = new THREE.MeshPhongMaterial({color: 0x0000FF});
  var malla = new THREE.Mesh(geometria, material);
  escena.add(malla);
  malla.rotation.set(10, 10, 0);
  var luz = new THREE.SpotLight(0xFFFFFF);
  luz.position.set(50, 50, 150);
  escena.add(luz);
  renderer.render(escena, camara);
}
window.addEventListener("load", iniciar);
```

Listado 12-4: *Agregando luz a la escena*

Además de la luz direccional agregada al final, hemos introducido algunos cambios más en el código del Listado 12-4 con respecto al ejemplo anterior. La configuración del renderer ahora incluye una segunda propiedad llamada **`antialias`** con el valor **`true`**. Esto suaviza la imagen en la pantalla y crea un efecto más realista. El material se define como **`MeshPhongMaterial()`** en color azul. El material Phong es probablemente el material más realista, pero también el que consume más recursos del ordenador. Es perfecto para nuestro ejemplo, pero siempre debemos considerar la posibilidad de alternar con otros materiales para lograr un balance entre calidad y desempeño.

El cubo se crea en el origen del mundo (las coordenadas `0`, `0`, `0`) y enfrenta a la cámara desde una de sus caras. Si graficamos la malla en esta posición, solo veremos un cuadrado en la pantalla. Para inicializar el cubo en una mejor posición, aplicamos una transformación usando la propiedad **`rotation`** y el método **`set()`**.

Para la luz usamos el constructor **`SpotLight()`** con un color blanco. Debido a que esta luz tiene una posición específica, tenemos que moverla a la posición adecuada. Esto se realiza mediante la propiedad **`position`** y el método **`set()`**.

Figura 12-5: *Luces reflejadas en un cubo*

Hágalo usted mismo: copie el código JavaScript del Listado 12-4 dentro del archivo webgl.js creado para el ejemplo anterior y abra el documento del Listado 12-1 en su navegador. Debería ver algo parecido a la Figura 12-5.

Texturas

Los colores y luces producen un efecto realista, pero el nivel de detalles en el mundo real es mucho más elevado: protuberancias, partículas, tejidos e incluso pequeñas moléculas en la superficie de los objetos producen millones de efectos visuales. Los detalles en el mundo real son tan complejos que reproducir un objeto de forma realista requeriría recursos que no se encuentran disponibles en los ordenadores actuales. Con la intención de solucionar este problema, los motores 3D ofrecen la posibilidad de agregar texturas a las mallas. Las texturas son imágenes que se dibujan en la superficie de una figura para simular detalles complejos.

Las texturas en Three.js se crean mediante un objeto de tipo `Texture` y luego se declaran como parte del material de la malla. La librería incluye el siguiente constructor para crear estos objetos.

Texture(imagen, mapeado, wrapS, wrapT, magFilter, minFilter**)**—Este constructor devuelve un objeto que representa la textura que se debe aplicar al material de la malla. El atributo **imagen** es la imagen para la textura. Se puede declarar como una imagen, un elemento **`<canvas>`** o un vídeo. El atributo **mapeado** define cómo se mapeará la textura sobre la superficie de la malla, los atributos **wrapS** y **wrapT** determinan cómo se distribuirá la imagen en la superficie y los atributos **magFilter** y **minFilter** declaran los filtros a aplicar en la textura para suavizar la imagen y producir un efecto más realista.

En el siguiente ejemplo, agregamos una textura al cubo para simular una caja de madera.

```
var canvas, imagen, renderer, escena, camara, malla;
function iniciar() {
  canvas = document.getElementById("canvas");
  imagen = document.createElement("img");
  imagen.src = "caja.jpg";
  imagen.addEventListener("load", crearmundo);
}
function crearmundo() {
  var ancho = canvas.width;
  var altura = canvas.height;
  renderer = new THREE.WebGLRenderer({canvas: canvas, antialias: true});
  renderer.setClearColor(0xFFFFFF);
  escena = new THREE.Scene();
  camara = new THREE.PerspectiveCamera(45, ancho / altura, 0.1, 1000);
  camara.position.set(0, 0, 150);
  var geometria = new THREE.BoxGeometry(50, 50, 50);
  var textura = new THREE.Texture(imagen);
  textura.needsUpdate = true;
  var material = new THREE.MeshPhongMaterial({map: textura});
  malla = new THREE.Mesh(geometria, material);
  escena.add(malla);
  var luz = new THREE.SpotLight(0xFFFFFF, 1);
  luz.position.set(0, 100, 250);
  escena.add(luz);
```

```
  renderer.render(escena, camara);
  canvas.addEventListener("mousemove", mover);
}
function mover(evento){
  malla.rotation.z = -evento.pageX * 0.01;
  malla.rotation.x = evento.pageY * 0.01;
  renderer.render(escena, camara);
}
window.addEventListener("load", iniciar);
```

Listado 12-5: *Agregando una textura a un objeto*

El archivo que contiene la imagen para la textura se tiene que descargar por completo antes de que se pueda aplicar al material. En la mayoría de las aplicaciones, se diseña una parte del código para descargar los recursos e indicar el progreso al usuario. En el ejemplo del Listado 12-5 hemos separado este proceso de la creación del mundo 3D para mostrar cómo se debería organizar el código. Más adelante en este capítulo estudiaremos otras alternativas mejores.

La función **`iniciar()`** crea un objeto **`<img>`**, declara al archivo caja.jpg como la fuente de la imagen y asigna la función **`crearmundo()`** como la función a ejecutar por el evento **`load`** para continuar el proceso cuando se carga el archivo.

La función **`crearmundo()`** sigue el mismo procedimiento que hemos usado antes para crear el mundo 3D y todos sus elementos, pero en este ejemplo, la textura se define antes de que el material se aplique a la malla. El objeto **`Texture`** se genera mediante el constructor **`Texture()`** usando la referencia a la imagen que hemos cargado previamente. Una vez que la textura está lista, se almacena en la variable **`textura`** y se asigna al material como uno de sus atributos. Todo este proceso se llama *mapeo de textura* y, por esa razón, la propiedad a cargo de asignar la textura al material se llama **`map`**.

Hágalo usted mismo: copie el código JavaScript del Listado 12-5 dentro del archivo webgl.js. Descargue el archivo caja.jpg desde nuestro sitio web. Suba estos archivos y el documento del Listado 12-1 a su servidor o servidor local, y abra el documento en su navegador. Mueva el ratón para rotar la caja.

IMPORTANTE: debido a las restricciones de origen cruzado estudiadas en el capítulo anterior, los archivos para las texturas tienen que estar localizados en el mismo servidor que la aplicación. Para probar este ejemplo, tiene que subir los archivos a un servidor o a un servidor local.

Figura 12-6: *Cubo con textura*

El objeto **`Texture`** incluye algunas propiedades para definir y configurar la textura.

needsUpdate—Esta propiedad informa al renderer que debe actualizar la textura. Se requiere cada vez que se define una nueva textura o se efectúan cambios en la textura actual (ver el ejemplo del Listado 12-5).

repeat—Esta propiedad define cuántas veces se tiene que repetir la imagen de la textura en el mismo lado de la malla. La propiedad declara o devuelve un vértice con dos coordenadas, **`x`** e **`y`** para cada eje del plano. Esta propiedad requiere que los atributos **wrapS** y **wrapT** del constructor del objeto **`Texture`** se declaren con el valor **`THREE.RepeatWrapping`** y el tamaño de la imagen usada para la textura tiene que tener una potencia de 2 (por ejemplo, 128 x 128, 256 x 256, 512 x 512, 1024 x 1024, etc.).

offset—Esta propiedad desplaza la imagen en la superficie y permite además de otros efectos, la creación de una animación simple pero efectiva. La propiedad declara o devuelve un vértice con dos coordenadas **`x`** e **`y`** para cada eje del plano. Esta propiedad requiere que los atributos **wrapS** y **wrapT** del constructor del objeto **`Texture`** se declaren con el valor **`THREE.RepeatWrapping`** y el tamaño de la imagen usada para la textura tiene que ser de una potencia de 2 (por ejemplo, 128 x 128, 256 x 256, 512 x 512, 1024 x 1024, etc.).

Mapeado UV

El mapeado UV es un proceso por el cual los puntos de la imagen de la textura se asocian con los vértices de la malla para dibujar la imagen en la posición exacta. El nombre deriva de los nombres asignados a los ejes de la imagen, U correspondiente a **x** y V correspondiente a **y** (**x** e **y** ya se usan para describir los ejes de los vértices de la malla).

Un efecto interesante logrado por el mapeado UV es la aplicación de diferentes materiales y, por lo tanto, texturas, a la misma geometría. El procedimiento se reserva normalmente para mallas complejas creadas por programas de modelado 3D, pero en el caso del cubo, Three.js ofrece una forma simple de hacerlo. El siguiente ejemplo aprovecha esta característica para convertir a nuestro cubo en un dado.

```
var renderer, escena, camara, malla;
function iniciar() {
  canvas = document.getElementById("canvas");
  var ancho = canvas.width;
  var altura = canvas.height;
  renderer = new THREE.WebGLRenderer({canvas: canvas, antialias: true});
  renderer.setClearColor(0xFFFFFF);
  escena = new THREE.Scene();
  camara = new THREE.PerspectiveCamera(45, ancho / altura, 0.1, 1000);
  camara.position.set(0, 0, 150);

  var materiales = [
    new THREE.MeshPhongMaterial({map:
THREE.ImageUtils.loadTexture("dado3.jpg")}),
    new THREE.MeshPhongMaterial({map:
THREE.ImageUtils.loadTexture("dado4.jpg")}),
    new THREE.MeshPhongMaterial({map:
THREE.ImageUtils.loadTexture("dado5.jpg")}),
    new THREE.MeshPhongMaterial({map:
THREE.ImageUtils.loadTexture("dado2.jpg")}),
```

```
    new THREE.MeshPhongMaterial({map:
THREE.ImageUtils.loadTexture("dado1.jpg")}),
    new THREE.MeshPhongMaterial({map:
THREE.ImageUtils.loadTexture("dado6.jpg")})
  ];
  var geometria = new THREE.BoxGeometry(50, 50, 50, 1, 1, 1);
  malla = new THREE.Mesh(geometria, new
THREE.MeshFaceMaterial(materiales));
  escena.add(malla);
  var luz = new THREE.SpotLight(0xFFFFFF, 2);
  luz.position.set(0, 100, 250);
  escena.add(luz);
  renderer.render(escena, camara);
  canvas.addEventListener("mousemove", mover);
}
function mover(evento){
  malla.rotation.x = evento.pageY * 0.01;
  malla.rotation.z = -evento.pageX * 0.01;
  renderer.render(escena, camara);
}
window.addEventListener("load", iniciar);
```

Listado 12-6: *Aplicando una textura diferente para cada lado del cubo*

En este ejemplo, aprovechamos el método **`loadTexture()`** del objeto **`ImageUtils`** provisto por la librería para descargar archivos y asignar las texturas al material sin interrumpir el resto del proceso. La malla y el mundo 3D completo se generan, y las texturas se aplican posteriormente cuando las imágenes terminan de cargarse. Esta es una forma sencilla de trabajar con texturas cuando la aplicación no requiere procesos más elaborados. El método **`loadTexture()`** descarga la imagen y devuelve el objeto **`Texture`** correspondiente, todo en un solo paso. El método nos evita tener que crear una función para descargar los recursos, pero debido a que no provee un buen mecanismo para controlar el proceso, no se recomienda en aplicaciones profesionales (lo incluimos en este código para simplificar el ejemplo).

Usando el método **`loadTexture()`** y el constructor **`MeshPhongMaterial()`** creamos un array con seis objetos **`Material`** diferentes, cada uno con su correspondiente textura. Este array se asigna más adelante como el segundo parámetro del constructor **`Mesh()`** usando el constructor **`MeshFaceMaterial()`**. Este es un constructor que toma un array y devuelve un tipo de material compuesto para la aplicación de diferentes materiales a la misma malla.

Es importante el orden de los materiales y texturas que se declaran en el array porque determina la ubicación exacta en la que se tienen que dibujar las imágenes en la superficie de la malla. En nuestro ejemplo, esto significa que las caras del dado se encontrarán en el lugar adecuado.

Figura 12-7: *Cubo convertido en un dado*

Hágalo usted mismo: copie el código JavaScript del Listado 12-6 dentro del archivo webgl.js. Descargue los archivos dado1.jpg, dado2.jpg, dado3.jpg, dado4.jpg, dado5.jpg, y dado6.jpg desde nuestro sitio web. Suba los archivos y el documento del Listado 12-1 a su servidor o servidor local, y abra el documento en su navegador. Mueva el ratón para girar el dado.

IMPORTANTE: aplicar texturas a figuras complejas requiere de un profundo conocimiento de mapeado UV y mapeado de texturas en general. Para obtener más información, visite nuestro sitio web y siga los enlaces de este capítulo.

Texturas de lienzo

Existen diferentes técnicas que podemos aplicar para crear efectos con las texturas, pero probablemente la más interesante sea usar un segundo elemento **<canvas>**. Esta alternativa nos permite acceder a una variedad de efectos provistos por la API Canvas para generar la textura dinámicamente, incluida la posibilidad de agregar texto 3D a nuestra escena.

```
var renderer, escena, camara, malla;
function iniciar() {
  canvas = document.getElementById("canvas");
  var ancho = canvas.width;
  var altura = canvas.height;
  renderer = new THREE.WebGLRenderer({canvas: canvas, antialias:true});
  renderer.setClearColor(0xCCFFFF);
  escena = new THREE.Scene();
  camara = new THREE.PerspectiveCamera(45, ancho / altura, 0.1, 1000);
  camara.position.set(0, 0, 150);
  var textocanvas = document.createElement("canvas");
  textocanvas.width = 512;
  textocanvas.height = 256;
  var contexto = textocanvas.getContext("2d");
  contexto.fillStyle = "rgba(255,0,0,0.95)";
  contexto.font = "bold 70px verdana, sans-serif";
  contexto.fillText("Texto en 3D", 0, 60);
  var geometria = new THREE.PlaneGeometry(100, 40);
  var textura = new THREE.Texture(textocanvas);
  textura.needsUpdate = true;
  var material = new THREE.MeshPhongMaterial({map: textura,
transparent: true, side: THREE.DoubleSide});
  malla = new THREE.Mesh(geometria, material);
  escena.add(malla);
  var luz = new THREE.PointLight(0xffffff);
  luz.position.set(0, 100, 250);
  escena.add(luz);
  renderer.render(escena, camara);
  canvas.addEventListener("mousemove", mover);
}
function mover(evento){
  malla.rotation.y = evento.pageX * 0.02;
  renderer.render(escena, camara);
}
window.addEventListener("load", iniciar);
```

Listado 12-7: *Incluyendo texto 3D en nuestra escena*

El lienzo para la textura es un segundo elemento **<canvas>** que no se mostrará en pantalla. Su único propósito es generar la imagen para la textura. Podríamos haber declarado este elemento en el documento y usar CSS para ocultarlo cambiando el valor de su propiedad **visibility**, pero la mejor manera de incluir este elemento es crear el objeto **Element** de forma dinámica con el método **createElement()**, como hemos hecho anteriormente con imágenes. En el Listado 12-7, el segundo elemento **<canvas>** se crea con este método y se definen sus atributos **width** y **height**. El proceso para dibujar un texto en el lienzo es el mismo que hemos explicado en el capítulo anterior: se crea el contexto 2D, se define el estilo para el texto y se dibuja la cadena de caracteres mediante el método **fillText()**.

El elemento **<canvas>** vuelve transparente las partes de la superficie del dibujo que no se están usando. Para mostrar el efecto producido, declaramos un color de fondo para el renderer con el método **setClearColor()** y aplicamos la textura a una geometría de plano. Esta es una malla plana con solo dos lados, como un cuadrado tridimensional, creada por el constructor **PlaneGeometry()**.

El resto del proceso es el mismo que hemos usado anteriormente, pero agregamos un nuevo parámetro al constructor **MeshPhongMaterial()** para declarar el material como un material de doble lado. Esto significa que la textura se mostrará al frente y al reverso de la malla.

Figura 12-8: *Texto 3D en el lienzo*

Hágalo usted mismo: copie el código JavaScript del Listado 12-7 dentro del archivo webgl.js y abra el documento del Listado 12-1 en su navegador. Mueva el ratón sobre el lienzo para rotar la malla y ver la textura en ambos lados.

Texturas de vídeo

Probablemente el efecto más sorprendente que podemos lograr aplicando texturas sea la inclusión de vídeos en nuestra escena. Esto es tan sencillo como asignar la referencia a un elemento **<video>** al constructor **Texture()**.

```
var canvas, video, renderer, escena, camara, malla;
function iniciar() {
  canvas = document.getElementById("canvas");
  video = document.createElement("video");
  video.src = "trailer.ogg";
  video.addEventListener("canplaythrough", crearmundo);
}
function crearmundo() {
  var ancho = canvas.width;
  var altura = canvas.height;
  renderer = new THREE.WebGLRenderer({canvas: canvas, antialias:true});
  renderer.setClearColor(0xFFFFFF);
  escena = new THREE.Scene();
  camara = new THREE.PerspectiveCamera( 45, ancho / altura, 0.1, 1000);
  camara.position.set(0, 0, 250);
```

```
  textura = new THREE.Texture(video);
  textura.minFilter = THREE.LinearFilter;
  textura.magFilter = THREE.LinearFilter;
  textura.generateMipmaps = false;

  var material = new THREE.MeshPhongMaterial({map: textura, side:
THREE.DoubleSide});
  var geometria = new THREE.PlaneGeometry(240, 135);
  malla = new THREE.Mesh(geometria, material);
  escena.add(malla);

  var luz = new THREE.PointLight(0xffffff);
  luz.position.set(0, 100, 250);
  escena.add(luz);

  canvas.addEventListener("mousemove", mover);
  video.play();
  graficar();
}
function mover(evento) {
  malla.rotation.y = event.pageX * 0.02;
}
function graficar() {
  textura.needsUpdate = true;
  renderer.render(escena, camara);
  requestAnimationFrame(graficar);
}
window.addEventListener("load", iniciar);
```

Listado 12-8: *Introduciendo video en un mundo 3D*

El código comienza con la creación del elemento **<video>**. La fuente del elemento se declara como el archivo trailer.ogg (el mismo archivo usado en los ejemplos del Capítulo 8). Para saber cuándo está listo el vídeo para ser reproducido, se agrega un listener para el evento **canplaythrough**. Cuando el navegador considera que tiene datos suficientes para reproducir el vídeo, el evento se desencadena y se ejecuta la función **crearmundo()**.

El procedimiento para la creación del mundo sigue los mismos pasos de los ejemplos anteriores, pero esta vez tenemos que configurar el objeto **Texture** para poder usar el vídeo como textura. Por defecto, Three.js aplica un filtro mipmap a las texturas. Este filtro produce un efecto antialias para suavizar la textura y volverla más realista, pero las texturas de vídeo no lo admiten. Después de la creación del objeto **Texture**, declaramos las propiedades **minFilter** y **magFilter** con el valor **THREE.LinearFilter** (un filtro simple) y desactivamos la generación de mipmaps.

Nuestro trabajo no termina aquí. Aún tenemos que crear el bucle que actualizará la textura por cada cuadro del vídeo y graficará nuevamente la escena en cada ciclo. Esto se realiza mediante la función **graficar()**. En esta función, la propiedad **needsUpdate** de la textura se declara como **true** y la escena se grafica nuevamente.

Al final de la función **crearmundo()**, se agrega un listener para el evento **mousemove** para que el usuario pueda interactuar con la malla. También comenzamos a reproducir el vídeo con el método **play()** y llamamos a la función **graficar()** por primera vez para iniciar el bucle.

Figura 12-9: *Textura de vídeo*
© Derechos Reservados 2008, Blender Foundation / www.bigbuckbunny.org

Hágalo usted mismo: copie el código JavaScript del Listado 12-8 dentro del archivo webgl.js. Descargue el archivo trailer.ogg de nuestro sitio web. Suba los archivos y el documento del Listado 12-1 a su servidor y abra el documento en su navegador. Mueva el ratón para rotar el vídeo.

IMPORTANTE: los vídeos OGG solo se reproducen en navegadores que admiten el formato OGG, como Google Chrome, Mozilla Firefox u Opera. Vea el Capítulo 8 para obtener más información.

Modelos 3D

Las mallas complejas son casi imposibles de desarrollar declarando los vértices individualmente o usando los constructores para figuras primitivas que hemos estudiado antes. El trabajo requiere programas 3D profesionales capaces de construir modelos elaborados que luego se puedan cargar e implementar en animaciones 3D. Uno de los programas más populares en el mercado es Blender (www.blender.org), creado por la fundación Blender y distribuido de forma gratuita. Este programa ofrece todas las herramientas necesarias para crear modelos 3D profesionales y también varios formatos en los que se pueden exportar modelos. Los desarrolladores y colaboradores de la librería Three.js han construido varias librerías externas que ayudan a los programadores a cargar archivos en estos formatos y adaptarlos a lo que Three.js puede entender y procesar.

En este momento, ya contamos con cargadores disponibles para el formato COLLADA, el formato OBJ, y el formato JSON, entre otros. Estas son pequeñas librerías que tenemos que incluir en el documento junto con el archivo Three.js. El siguiente ejemplo incluye el archivo ColladaLoader.js con la librería COLLADA para poder procesar objetos en el formato COLLADA e introducir un coche de policía a nuestro mundo 3D.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Three.js</title>
  <script src="three.min.js"></script>
  <script src="ColladaLoader.js"></script>
```

```
  <script>
    var canvas, video, renderer, escena, camara, malla;
    function iniciar() {
      canvas = document.getElementById("canvas");
      var cargador = new THREE.ColladaLoader();
      cargador.load("police.dae", crearmundo);
    }
    function crearmundo(collada) {
      var ancho = canvas.width;
      var altura = canvas.height;
      renderer = new THREE.WebGLRenderer({canvas: canvas,
antialias:true});
      renderer.setClearColor(0xFFFFFF);
      escena = new THREE.Scene();
      camara = new THREE.PerspectiveCamera(45, ancho / altura, 0.1, 1000);
      camara.position.set(0, 0, 150);
      malla = collada.scene;
      malla.scale.set(20, 20, 20);
      malla.rotation.set(-Math.PI / 2, 0, 0);
      escena.add(malla);
      var luz = new THREE.PointLight(0xffffff, 2, 0);
      luz.position.set(0, 100, 250);
      escena.add(luz);
      renderer.render(escena, camara);
      canvas.addEventListener("mousemove", mover);
    }
    function mover(evento) {
      malla.rotation.z = -evento.pageX * 0.01;
      renderer.render(escena, camara);
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <canvas id="canvas" width="500" height="400"></canvas>
  </section>
</body>
</html>
```

Listado 12-9: *Cargando modelos 3D*

Los modelos exportados con COLLADA se almacenan en varios archivos. Después de exportar el modelo, tendremos un archivo de texto con la extensión .dae que contiene toda la especificación del modelo y uno o más archivos con las imágenes para las texturas. El modelo de nuestro ejemplo se almacena en el archivo police.dae y se incluyen dos archivos para las texturas (SFERIFF.JPG y SFERIFFI.JPG).

Los archivos de modelos se deben descargar como cualquier otro recurso, pero el cargador COLLADA facilita sus propios métodos para hacerlo. En la función **`iniciar()`** del Listado 12-9, el constructor **`ColladaLoader()`** se usa para crear el objeto **`Loader`** y se llama al método **`load()`** que ofrece este objeto para descargar el archivo .dae (solo se tiene que declarar el archivo .dae en el método, los archivos para las texturas se descargan de forma automática). El método **`load()`** tiene dos atributos, el atributo **archivo** para indicar la ruta del archivo a

descargar y el atributo **función** para declarar una función a la que se llamará cuando la descarga del archivo haya finalizado. El método `load()` envía un objeto `Scene` a esta función que podemos procesar para insertar el modelo a nuestra propia escena.

En nuestro ejemplo, la función que procesa el objeto `Scene` es `crearmundo()`. Después de seguir el procedimiento estándar para crear el renderer, la escena y la cámara, el objeto `Scene` que representa el modelo se almacena en la variable `malla`, se escala a las dimensiones de nuestro mundo, se rota al ángulo correcto para enfrentar la cámara y finalmente se agrega a la escena.

Figura 12-10: *Modelo COLLADA*
Modelo provisto por TurboSquid Inc. (www.turbosquid.com)

Hágalo usted mismo: el archivo para el cargador COLLADA está disponible en el paquete Three.js dentro del directorio examples. Siga la ruta examples/js/loaders/ y copie el archivo ColladaLoader.js en el directorio de su proyecto. Este archivo tiene que incluirse en el documento como lo hemos hecho en el Listado 12-9 para poder acceder a los métodos que nos permiten leer y procesar archivos en este formato. Cree un nuevo archivo HTML con el documento del Listado 12-9, suba este archivo, el documento HTML, y los tres archivos del modelo a su servidor o servidor local, y abra el documento en su navegador.

IMPORTANTE: el modelo usado en este ejemplo se acompaña de dos archivos que contienen las imágenes para las texturas (SFERIFF.JPG y SFERIFFI.JPG). Los tres archivos están disponibles en nuestro sitio web.

Animaciones 3D

Las animaciones en 3D solo difieren de las animaciones en 2D en cómo se construyen los cuadros. En Three.js, el renderer hace la mayoría del trabajo y todo el proceso es casi automático, mientras que en un contexto en 2D, tenemos que limpiar la superficie en cada ciclo nosotros mismos. A pesar de las pequeñas diferencias, los requerimientos del código para una aplicación profesional son siempre los mismos. Cuanto mayor es la complejidad, mayor es la necesidad de crear una organización apropiada y la mejor manera de hacerlo es trabajando con propiedades y métodos declarados en un objeto global, como lo hemos hecho en el Capítulo 11.

Para ofrecer un ejemplo de una animación en 3D, vamos a crear un pequeño videojuego. En el juego tenemos que conducir un automóvil en un área rodeada de muros. El propósito es capturar las esferas verdes que se encuentran flotando dentro de la habitación. El siguiente es el documento necesario para esta aplicación.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Three.js</title>
  <style>
    body {
      margin: 0px;
      overflow: hidden;
    }
  </style>
  <script src="three.min.js"></script>
  <script src="ColladaLoader.js"></script>
  <script src="webgl.js"></script>
</head>
<body></body>
</html>
```

Listado 12-10: *Creando del documento para un videojuego en 3D*

El documento es sencillo. Solo los estilos CSS para el elemento **<body>** pueden resultar extraños. Como vamos a usar toda la ventana del navegador para el renderer, se requieren estas propiedades para asegurarnos de que ningún margen o barra de desplazamiento ocupa el espacio que necesitamos para nuestra aplicación.

Los archivos JavaScript incluidos son el archivo three.min.js de la librería Three.js, el archivo ColladaLoader.js para volver a incorporar a la escena nuestro coche de policía, y el archivo webgl.js con el código de nuestro juego.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 12-10 y copie los códigos JavaScript presentados a continuación dentro de un archivo vacío llamado webgl.js. Todos los códigos estudiados de aquí en adelante son necesarios para ejecutar la aplicación.

En general, el código para una aplicación 3D es más extenso que el código JavaScript que se requiere para otros propósitos. Para simplificar este ejemplo, primero vamos a presentar el objeto global y sus propiedades esenciales, y agregaremos el resto de las propiedades y métodos necesarios para el juego en los siguientes listados.

```
var mijuego = {
  renderer: "",
  escena: "",
  camara: "",
  luz: "",
  coche: {
    malla: "",
    velocidad: 0,
    incrementar: false,
    izquierda: false,
    derecha: false,
    angulorueda: 0
  },
```

```
  muros: [{x: 0, y: 100, z: -1000},
          {x: -1000, y: 100, z: 0},
          {x: 0, y: 100, z: 1000},
          {x: 1000, y: 100, z:0}],

texturas: {
    coche: "",
    piso: "",
    muros: ""
  },
  objetivos: {
    malla: "",
    limites: [],
  },
  entrada: []
};
```

Listado 12-11: *Definiendo propiedades básicas*

El objeto global para esta aplicación se llama **mijuego**. Comenzamos la definición de **mijuego** declarando las propiedades necesarias para controlar y modificar el estado de nuestro juego. Las propiedades **renderer**, **escena**, **camara** y **luz** almacenan las referencias en los elementos de nuestro mundo 3D. La propiedad **coche** contiene un objeto con los datos básicos de nuestro coche, como la malla y el valor de la velocidad actual. La propiedad **muros** es un array que contiene cuatro vértices para la definición de la posición de los muros. La propiedad **texturas** es un objeto con propiedades que referencian las texturas del coche, el suelo y los muros. La propiedad **objetivos** es también un objeto que almacena la malla y los límites de las esferas verdes (el objetivo de nuestro juego). Finalmente, la propiedad **entrada** es un array vacío que usaremos para almacenar las teclas que pulsa el usuario.

Algunas de estas propiedades se inicializan con valores vacíos. El siguiente paso es declarar los valores correctos y definir los elementos principales de nuestra escena cuando se carga el código. Como siempre, esto se realiza en nuestro método **iniciar()**.

```
mijuego.iniciar = function() {
  var ancho = window.innerWidth;
  var altura = window.innerHeight;
  var canvas = document.createElement("canvas");
  canvas.width = ancho;
  canvas.height = altura;
  document.body.appendChild(canvas);

  mijuego.renderer = new THREE.WebGLRenderer({canvas: canvas,
antialias:true});
  mijuego.escena = new THREE.Scene();
  mijuego.camara = new THREE.PerspectiveCamera(45, ancho / altura, 0.1,
10000);
  mijuego.camara.position.set(0, 50, 150);
  mijuego.luz = new THREE.PointLight(0x999999, 1);
  mijuego.luz.position.set(0, 50, 150);
  mijuego.escena.add(mijuego.luz);

  window.addEventListener("keydown", function(evento) {
    mijuego.entrada.push({type: "keydown", key: evento.key});
  });
```

```
  window.addEventListener("keyup", function(evento) {
    mijuego.entrada.push({type: "keyup", key: evento.key});
  });
  mijuego.cargar();
  mijuego.crear();
};
```

Listado 12-12: *Definiendo el método* `iniciar()`

Para nuestro juego queremos usar la ventana completa del navegador, pero las dimensiones de este espacio varían de acuerdo a cada dispositivo o la configuración establecida por el usuario. Para crear un elemento **<canvas>** tan grande como la ventana, lo primero que tenemos que hacer es obtener las dimensiones actuales de la ventana leyendo las propiedades **innerWidth** e **innerHeight** del objeto **Window** (ver Capítulo 6). Usando estos valores, el lienzo se crea dinámicamente, se dimensiona y se agrega al cuerpo mediante el método **appendChild()**.

Después de las usuales definiciones del renderer, la escena, la cámara y la luz, se agregan dos listeners a la ventana para los eventos **keydown** y **keyup**. Estos listeners son necesarios para controlar la entrada del usuario. Se desencadena el evento **keydown** cuando se pulsa una tecla y se desencadena el evento **keyup** cuando se libera una tecla. Ambos envían un objeto **KeyboardEvent** a la función con la propiedad **key** que contiene un valor que identifica la techa que ha desencadenado el evento (ver Capítulo 6, Listado 6-165). Se han declarado dos funciones anónimas para procesar estos eventos y almacenar el valor de la propiedad **key** en el array **entrada** cuando se desencadenan. Procesaremos esta entrada más adelante.

Al final del método **iniciar()**, se llama al método **cargar()** para descargar los archivos con la descripción del modelo y las texturas.

```
mijuego.cargar = function() {
  var cargador = new THREE.ColladaLoader();
  cargador.load("police.dae", function(collada) {
    mijuego.texturas.coche = collada;
  });
  var imagen1 = document.createElement("img");
  imagen1.src = "asfalto.jpg";
  imagen1.addEventListener("load", function(evento) {
    mijuego.texturas.piso = evento.target;
  });
  var imagen2 = document.createElement("img");
  imagen2.src = "muro.jpg";
  imagen2.addEventListener("load", function(evento) {
    mijuego.texturas.muros = evento.target;
  });
  var controlbucle = function() {
    if (mijuego.texturas.coche && mijuego.texturas.piso &&
mijuego.texturas.muros) {
      mijuego.crear();
    } else {
      setTimeout(controlbucle, 200);
    }
  };
  controlbucle();
};
```

Listado 12-13: *Definiendo el método* `cargar()`

Este método es un cargador pequeño pero práctico. El cargador comienza el proceso de descarga para cada uno de los archivos (el modelo COLLADA, y las texturas para el piso y los muros) y define una pequeña función interna que controla el proceso. Cuando los archivos terminan de cargarse, los objetos obtenidos se almacenan en las propiedades correspondientes (**`texturas.coche`**, **`texturas.piso`** y **`texturas.muros`**). La función **`controlbucle()`** se llama a sí misma y genera un bucle que controla constantemente los valores de estas propiedades y ejecuta el método **`crear()`** solo cuando el modelo y ambas texturas terminan de cargarse.

```
mijuego.crear = function() {
  var geometria, material, textura, malla;
  malla = mijuego.texturas.coche.scene;
  malla.scale.set(20, 20, 20);
  malla.rotation.set(-Math.PI / 2, 0, Math.PI);
  malla.position.y += 14;
  mijuego.escena.add(malla);
  mijuego.coche.malla = malla;

  geometria = new THREE.PlaneGeometry(2000, 2000, 10, 10);
  textura = new THREE.Texture(mijuego.texturas.piso, THREE.UVMapping,
THREE.RepeatWrapping, THREE.RepeatWrapping);
  textura.repeat.set(20, 20);
  textura.needsUpdate = true;

  material = new THREE.MeshPhongMaterial({map: textura});
  malla = new THREE.Mesh(geometria, material);
  malla.rotation.x = Math.PI * 1.5;
  mijuego.escena.add(malla);

  for (var f = 0; f < 4; f++) {
    geometria = new THREE.PlaneGeometry(2000, 200, 10, 10);
    textura = new THREE.Texture(mijuego.texturas.muros,
THREE.UVMapping, THREE.RepeatWrapping, THREE.RepeatWrapping);
    textura.repeat.set(10, 1);
    textura.needsUpdate = true;

    material = new THREE.MeshPhongMaterial({map: textura});
    malla = new THREE.Mesh(geometria, material);
    malla.position.set(mijuego.muros[f].x, mijuego.muros[f].y,
mijuego.muros[f].z);
    malla.rotation.y = Math.PI / 2 * f;
    mijuego.escena.add(malla);
  }
  mijuego.bucle();
};
```

***Listado 12-14:** Definiendo el método* `crear()`

En el método **`crear()`** se crean las mallas para el coche, el piso y los cuatro muros. No incluimos nada nuevo en este método excepto el uso de la propiedad **`repeat`**. Esta propiedad declara cuántas veces se tiene que repetir la imagen en el lado de la malla para crear la textura. Esto es necesario debido a que la imagen para la textura de los muros es un cuadrado, pero los muros son 10 veces más largos que altos. Si declaramos los valores de la propiedad **`repeat`** como `10, 1` (**`textura.repeat.set(10, 1);`**), la imagen se dibuja en los muros en las proporciones adecuadas.

Como hemos mencionado antes, para que la propiedad **`repeat`** trabaje adecuadamente, se deben satisfacer algunas condiciones. El tamaño de la imagen para la textura tiene que ser una potencia de 2 (por ejemplo, 128 x 128, 256 x 256, 512 x 512, 1024 x 1024, etc.), se debe declarar más de un segmento para cada plano de la geometría, y los atributos **wrapS** y **wrapT** del constructor **`Texture()`** se tienen que declarar con el valor **`THREE.RepeatWrapping`**. Estos atributos se declaran por defecto con el valor **`THREE.ClampToEdgeWrapping`**, lo que significa que la imagen se escalará para ocupar todo el lado de la malla (como ha pasado en ejemplos anteriores). La constante **`THREE.RepeatWrapping`** cancela este efecto y nos permite distribuir la imagen del modo que queramos.

La definición del mundo 3D ha finalizado. Ahora, es el momento de programar los métodos que controlarán el proceso principal. Necesitamos un método para controlar la entrada del usuario, otro método para calcular la nueva posición del coche, un tercer método para detectar colisiones y uno más para actualizar el renderer.

```
mijuego.control = function(evento) {
  var accion;
  while (mijuego.entrada.length) {
    accion = mijuego.entrada.shift();
    switch (accion.type) {
      case "keydown":
        switch(accion.key){
          case "ArrowUp":
            mijuego.coche.incrementar = true;
            break;
          case "ArrowLeft":
            mijuego.coche.izquierda = true;
            break;
          case "ArrowRight":
            mijuego.coche.derecha = true;
            break;
        }
        break;
      case "keyup":
        switch(accion.key){
          case "ArrowUp":
            mijuego.coche.incrementar = false;
            break;
          case "ArrowLeft":
            mijuego.coche.izquierda = false;
            break;
          case "ArrowRight":
            mijuego.coche.derecha = false;
            break;
        }
        break;
    }
  }
};
```

Listado 12-15: *Definiendo el método* `control()`

El método **`control()`** del Listado 12-15 procesa los valores que se han almacenado en el array **`entrada`** con las funciones que responden a los eventos **`keydown`** y **`keyup`**. Esto es

parte de una técnica que se usa para evitar el retraso que genera el sistema cada vez que se pulsa una tecla. Cuando el usuario pulsa o libera una tecla, la acción la detectan los eventos **keydown** y **keyup**, y la nueva condición se almacena en las propiedades **incrementar**, **izquierda** o **derecha** respectivamente. Estas propiedades devuelven **true** desde el momento que se pulsa la tecla y hasta que se libera. Al usar este procedimiento, no existe ningún retraso y el coche responde instantáneamente.

El valor de la propiedad **key** que se toma del array **entrada** se compara con los textos "ArrowUp" (flecha arriba), "ArrowLeft" (flecha izquierda) y "ArrowRight" (flecha derecha) para identificar qué tecla ha pulsado o liberado el usuario. Estos textos se corresponden con las teclas flecha arriba, flecha izquierda y flecha derecha, respectivamente (ver el objeto **KeyboardEvent** en el Capítulo 6). De acuerdo con estos valores y el evento al que estamos respondiendo, las propiedades **incrementar**, **izquierda** y **derecha** se declaran con los valores **true** o **false** para indicar la condición actual al resto de los métodos del código.

IMPORTANTE: se puede usar un sistema de entrada similar al que programamos para esta aplicación con el fin de almacenar cualquier tipo de entrada, no solo teclas. Por ejemplo, podemos almacenar valores personalizados en el array **entrada** con los que representar los eventos del ratón y luego responder a estos valores desde el método **control()** de la misma forma que lo hemos hecho en este ejemplo para las teclas de las flechas.

Las propiedades **incrementar**, **izquierda** y **derecha** nos ayudan a definir la velocidad y rotación del coche. Todo lo demás se calcula con esta información, desde la posición de la cámara hasta la detección de colisiones contra los muros o las esferas. Una gran parte de este trabajo se realiza mediante el método **procesar()**.

```
mijuego.procesar = function() {
  if (mijuego.coche.incrementar) {
    if (mijuego.coche.velocidad < 8) {
      mijuego.coche.velocidad += 0.1;
    }
  } else {
    if (mijuego.coche.velocidad > 0) {
      mijuego.coche.velocidad -= 0.1;
    } else {
      mijuego.coche.velocidad += 0.1;
    }
  }
  if (mijuego.coche.izquierda && mijuego.coche.angulorueda > - 0.5) {
    mijuego.coche.angulorueda -= 0.01;
  }
  if (!mijuego.coche.izquierda && mijuego.coche.angulorueda < 0) {
    mijuego.coche.angulorueda += 0.02;
  }
  if (mijuego.coche.derecha && mijuego.coche.angulorueda < 0.5) {
    mijuego.coche.angulorueda += 0.01;
  }
  if (!mijuego.coche.derecha && mijuego.coche.angulorueda > 0) {
    mijuego.coche.angulorueda -= 0.02;
  }
  var angulo = mijuego.coche.malla.rotation.z;
```

```
  angulo -= mijuego.coche.angulorueda * mijuego.coche.velocidad / 100;
  mijuego.coche.malla.rotation.z = angulo;
  mijuego.coche.malla.position.x += Math.sin(angulo) *
mijuego.coche.velocidad;
  mijuego.coche.malla.position.z += Math.cos(angulo) *
mijuego.coche.velocidad;

  var desviacion = mijuego.coche.angulorueda / 3;
  posx = mijuego.coche.malla.position.x -
(Math.sin(mijuego.coche.malla.rotation.z + desviacion) * 150);
  posz = mijuego.coche.malla.position.z -
(Math.cos(mijuego.coche.malla.rotation.z + desviacion) * 150);
  mijuego.camara.position.set(posx, 50, posz);
  mijuego.camara.lookAt(mijuego.coche.malla.position);
  mijuego.luz.position.set(posx, 50, posz);
};
```

Listado 12-16: *Definiendo el método* `procesar()`

Existen dos valores importantes que tenemos que recalcular en cada ciclo del bucle para procesar la entrada del usuario: la velocidad y el ángulo del coche. La velocidad se incrementa mientras el usuario mantiene pulsada la flecha hacia arriba y se reduce cuando se libera la tecla. La primera instrucción **`if else`** en el método **`procesar()`** del Listado 12-16 controla esta situación. Si el valor de **`incrementar`** es **`true`**, la velocidad del coche se incrementa en **`0.1`**, hasta un máximo de **`8`**. En caso contrario, la velocidad se reduce gradualmente a **`0`**.

Las siguientes cuatro instrucciones **`if`** controlan el ángulo de las ruedas. Este ángulo se suma o resta al ángulo del coche para girar hacia la izquierda o la derecha de acuerdo a los valores de las propiedades **`izquierda`** y **`derecha`**. Al igual que las instrucciones para la velocidad, las instrucciones **`if`** para el ángulo de las ruedas limitan los valores posibles (desde **`-0.5`** hasta **`0.5`**).

Una vez que se establecen los valores de la velocidad y el ángulo de las ruedas, comenzamos el proceso de calcular la nueva posición de cada uno de los elementos de la escena. Primero, el ángulo actual del coche se almacena en la variable **`angulo`** (este es el ángulo en el eje **`z`**). A continuación, se resta el ángulo de las ruedas del valor del ángulo del coche y este nuevo valor se asigna nuevamente a la propiedad **`rotation`** del coche para rotar la malla (**`mijuego.coche.malla.rotation.z = angulo`**).

Con el ángulo establecido, es hora de determinar la nueva posición del coche. Los valores para las coordenadas **`x`** y **`z`** se calculan desde el ángulo y la velocidad usando la fórmula **`Math.sin(angulo) × velocidad`** y **`Math.cos(angulo) × velocidad`**. Los resultados se asignan de inmediato a las propiedades **`position.x`** y **`position.z`** del coche, y la malla se mueve a la nueva posición, pero aún tenemos que mover la cámara y la luz, o nuestro vehículo pronto desaparecerá en las sombras. Usando los valores de la nueva posición del coche, su ángulo y la distancia permanente entre la cámara y el coche (**`150`**), se calculan los valores de las coordenadas de la cámara. Estos valores se almacenan en las variables **`posx`** y **`posy`** y se asignan a la propiedad **`position`** de la cámara y la luz (la cámara y la luz tienen siempre las mismas coordenadas). Con esta fórmula, la cámara da la impresión de estar unida a la parte posterior del coche. Para lograr un efecto más realista, se calcula un pequeño valor de desviación a partir del ángulo de las ruedas y se suma al ángulo del coche en la última fórmula, moviendo la cámara levemente hacia un lado u otro (**`desviacion = mijuego.coche.angulorueda / 3`**).

Al final del método **procesar()**, apuntamos la cámara a la nueva posición del coche mediante el método **lookAt()** y el vértice que devuelve la propiedad **position** (**mijuego.coche.malla.position**). Sin importar cuánto cambia el valor de esta propiedad, la cámara siempre apuntará al coche.

Lo básico: las fórmulas matemáticas implementadas en este ejemplo son funciones trigonométricas simples que se usan para obtener los valores de las coordenadas a partir de los ángulos y puntos en un sistema de coordenadas de dos dimensiones. Para estudiar otros ejemplos, vea el Capítulo 11, Listado 11-30.

Con las posiciones del coche, la cámara y la luz ya definidas, estamos listos para graficar la escena, solo nos queda un objeto por incluir. Las esferas verdes que tienen que perseguir nuestro coche se posicionan al azar en la escena, una cada vez. Cuando pasamos a través de la esfera actual, se crea una nueva posición diferente. Debido a que esta operación puede ocurrir en cualquier momento durante la animación, decidimos incluirla junto con el proceso de graficado en el método **dibujar()**.

```
mijuego.dibujar = function() {
  if (!mijuego.objetivos.malla) {
    var geometria = new THREE.SphereGeometry(20, 10, 10);
    var material = new THREE.MeshBasicMaterial({color: 0x00FF00,
wireframe: true});
    var malla = new THREE.Mesh(geometria, material);
    var posx = (Math.random() * 1800) - 900;
    var posz = (Math.random() * 1800) - 900;
    malla.position.set(posx, 30, posz);
    mijuego.escena.add(malla);

    mijuego.objetivos.malla = malla;
    mijuego.objetivos.limites[0] = posx - 30;
    mijuego.objetivos.limites[1] = posx + 30;
    mijuego.objetivos.limites[2] = posz - 30;
    mijuego.objetivos.limites[3] = posz + 30;
  } else {
    mijuego.objetivos.malla.rotation.y += 0.02;
  }
  mijuego.renderer.render(mijuego.escena, mijuego.camara);
};
```

Listado 12-17: *Definiendo el método* `dibujar()`

Si no se ha creado ninguna esfera, el método **dibujar()** genera una nueva. El material se declara como una malla de alambre verde y la posición se determina al azar con el método **random()**. Después de agregar la malla a la escena, se calcula un área alrededor de la esfera para poder detectar su posición más adelante.

Debido a que la posición de cada elemento en la escena se determina mediante solo un vértice, es casi imposible detectar una colisión entre los elementos comparando estas coordenadas. Al agregar y restar 30 unidades a cada coordenada de la esfera y almacenar esos valores en el array **objetivos.limites**, generamos un área virtual de 60 unidades de ancho que contiene la esfera (ver Figura 12-11 debajo). Cuando el vértice de la posición del coche se encuentra dentro de esta área, se confirma la colisión.

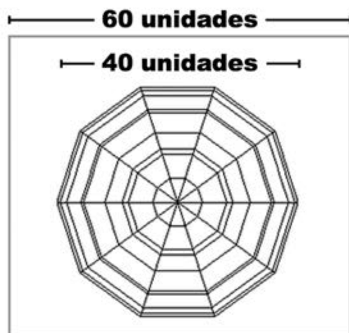

Figura 12-11: *Expandiendo el área ocupada por la esfera*

La última operación del método **dibujar()** del Listado 12-17 grafica la escena en el lienzo. Aunque esto es importante, no es lo último que tenemos que hacer. Aún tenemos que mejorar el sistema de detección de colisiones. El coche de nuestro juego puede chocar contra cualquiera de los cuatro muros y las esferas verdes. El método **detectar()** es el que se encarga de detectar y responder a estas situaciones.

```
mijuego.detectar = function() {
  var posx = mijuego.coche.malla.position.x;
  var posz = mijuego.coche.malla.position.z;
  if (posx < -980 || posx > 980 || posz < -980 || posz > 980) {
    mijuego.coche.velocidad = -7;
  }
  if (posx > mijuego.objetivos.limites[0] && posx <
mijuego.objetivos.limites[1] && posz > mijuego.objetivos.limites[2] &&
posz < mijuego.objetivos.limites[3]) {
    mijuego.escena.remove(mijuego.objetivos.malla);
    mijuego.objetivos.malla = "";
  }
};
```

Listado 12-18: *Definiendo el método* `detectar()`

Comparando las coordenadas actuales del coche con las posición de los muros y los límites del área que rodea a la esfera, determinamos las posibles colisiones. Si el vértice de la posición del coche cae fuera del área del juego (más allá de los muros), el valor de la propiedad **velocidad** se declara como -7, y hace que el coche rebote del muro. En el caso de que este vértice caiga dentro del área de la esfera, la malla se elimina del mundo 3D mediante el método **remove()** y se declara la propiedad **objetivos.malla** con un valor nulo. Cuando esto ocurre, la función **dibujar()** del Listado 12-17 detecta la ausencia de una esfera y dibuja una nueva.

La detección es lo último que necesitamos para nuestro juego, aunque aún tenemos que construir el bucle que llama a todos estos métodos una y otra vez.

```
mijuego.bucle = function() {
  mijuego.control();
  mijuego.procesar();
  mijuego.detectar();
  mijuego.dibujar();
```

```
  requestAnimationFrame(function() {
    mijuego.bucle();
  });
};
window.addEventListener("load", function() {
  mijuego.iniciar();
});
```

Listado 12-19: *Definiendo el método* `bucle()`

El código JavaScript de este ejemplo se ha simplificado con fines educativos. No tenemos un mensaje de *game over*, puntos que celebrar o vidas que perder. La buena noticia es que ya hemos estudiado todo lo que necesitamos saber para agregar estas características y finalizar nuestro juego.

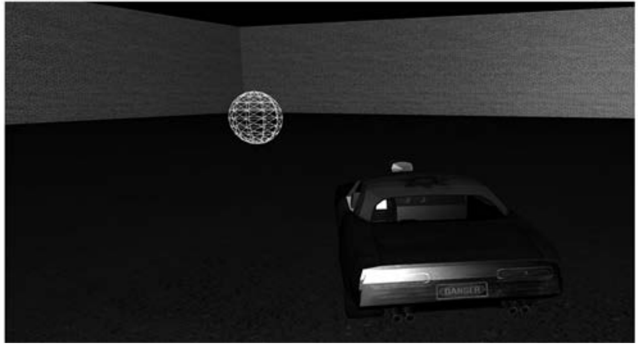

Figura 12-12: *Videojuego en 3D para la Web*
Modelos y texturas provistos por TurboSquid Inc. (www.turbosquid.com)

Hágalo usted mismo: todos los códigos presentados en esta parte del capítulo trabajan juntos para crear el videojuego, incluido el documento del Listado 12-10. Intente aplicar la API Fullscreen a este ejemplo para llevar el juego a pantalla completa.

IMPORTANTE: las mallas, las texturas y los modelos usados en los ejemplos de este capítulo tienen derechos registrados y no otorgan derechos de distribución. Consulte con TurboSquid Inc. (www.turbosquid.com) o la fundación Blender (www.blender.org) antes de usar este material en sus propios proyectos.

Capítulo 13

API Pointer Lock

13.1 Puntero personalizado

Las aplicaciones visuales, como las creadas con el elemento **<canvas>** o WebGL, a veces requieren el uso de métodos alternativos para expresar los movimientos del ratón. Existen incontables razones por las que se puede solicitar este requisito. Podríamos necesitar cambiar el gráfico que representa el puntero para indicar una función diferente para el ratón u ocultarlo por completo para evitar distraer al usuario de la imagen o el vídeo que le estamos mostrando. Considerando estos requerimientos, los navegadores incluyen la API Pointer Lock.

Captura del ratón

La API Pointer Lock es un grupo pequeño de propiedades, métodos y eventos que ayudan a la aplicación a tomar control sobre el ratón. Se facilitan los siguientes métodos para bloquear y desbloquear el ratón.

requestPointerLock()—Este método bloquea el ratón y lo vincula a un elemento. El método está disponible en los objetos **Element**.

exitPointerLock()—Este método desbloquea el ratón y vuelve visible el puntero. El método está disponible en el objeto **Document**.

Cuando se llama al método **requestPointerLock()**, el gráfico que representa el puntero (normalmente una pequeña flecha) se oculta y el código se vuelve responsable de facilitar la referencia visual que necesita el usuario para interactuar con la aplicación. El siguiente ejemplo ilustra cómo asumir el control del ratón.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Pointer Lock</title>
  <script>
    function iniciar() {
      var elemento = document.getElementById("control");
      elemento.addEventListener("click", bloquearraton);
    }
    function bloquearraton(evento) {
      var elemento = evento.target;
      elemento.requestPointerLock();
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
```

```
<body>
  <section>
    <p id="control">Clic aquí para bloquear el ratón</p>
  </section>
</body>
</html>
```

Listado 13-1: *Asumiendo control del ratón*

A menos que el elemento que solicita el control del ratón se encuentre en modo de pantalla completa, el método **`requestPointerLock()`** devolverá un error cuando se llame sin la intervención del usuario. Un gesto del usuario, como el clic del ratón, debe preceder a la acción. Considerando esta condición, el código del Listado 13-1 agrega un listener para el evento **`click`** al elemento **`<p>`**. Cuando el usuario hace clic en este elemento, se llama a la función **`bloquearraton()`** y se usa el método **`requestPointerLock()`** para bloquear el ratón. En este momento, el puntero desaparece de la pantalla y no se muestra nuevamente a menos que el usuario abra otra ventana o cancele el modo pulsando la tecla Escape del teclado.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 13-1. Abra el documento en su navegador y haga clic en el área ocupada por el elemento **`<p>`**. Debería ver desaparecer el puntero del ratón. Pulse la tecla Escape en su teclado para desbloquear el ratón.

Cuando se bloquea el ratón para un elemento, el resto de los elementos no desencadenan ningún evento del ratón hasta que se desbloquea por la aplicación o el usuario cancela el modo. Para informar de lo que ocurre, la API incluye estos eventos.

pointerlockchange—Este evento se desencadena en el objeto **`Document`** cuando un elemento bloquea o desbloquea el ratón.

pointerlockerror—Este evento se desencadena en el objeto **`Document`** cuando falla el intento de bloquear al ratón.

El código del siguiente ejemplo responde al evento **`pointerlockchange`** e imprime un mensaje en la consola cada vez que cambia la condición del ratón.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Pointer Lock</title>
  <script>
    function iniciar() {
      var elemento = document.getElementById("control");
      elemento.addEventListener("click", bloquearraton);
      document.addEventListener("pointerlockchange", controlarraton);
    }
    function bloquearraton(evento) {
      var elemento = evento.target;
      elemento.requestPointerLock();
    }
```

```
    function controlarraton() {
      console.log("La condición del ratón cambió");
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <p id="control">Clic aquí para bloquear el ratón</p>
  </section>
</body>
</html>
```

Listado 13-2: *Reportando un cambio en la condición del ratón*

Hágalo usted mismo: actualice el documento en su archivo HTML con el código del Listado 13-2. Abra el documento en su navegador y active la consola para ver los mensajes generados por el código. Cada vez que haga clic en el elemento o pulse la tecla Escape, se debería mostrar un mensaje debería en la consola.

El ratón se bloquea para un elemento específico. La API incluye la siguiente propiedad para informar de cuál es el elemento que ha bloqueado al ratón.

pointerLockElement—Esta propiedad devuelve una referencia al objeto **`Element`** que representa al elemento que ha bloqueado al ratón o el valor **`null`** si el ratón no se ha bloqueado.

La propiedad **`pointerLockElement`** se puede usar junto con el evento **`pointerlockchange`** para determinar si el ratón se ha bloqueado o desbloqueado.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Pointer Lock</title>
  <script>
    function iniciar() {
      var elemento = document.getElementById("control");
      elemento.addEventListener("click", bloquearraton);
      document.addEventListener("pointerlockchange", controlarraton);
    }
    function bloquearraton(evento) {
      var elemento = evento.target;
      elemento.requestPointerLock();
    }
    function controlarraton() {
      var elemento = document.getElementById("control");
      if (document.pointerLockElement) {
        console.log("Ratón Bloqueado");
      } else {
        console.log("Ratón Liberado");
      }
    }
```

```
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <p id="control">Clic aquí para bloquear el ratón</p>
  </section>
</body>
</html>
```

Listado 13-3: *Comprobando la condición del ratón*

Cada vez que el ratón se bloquea o desbloquea, mediante nuestro código o por el usuario, la función **`controlarraton()`** del Listado 13-3 lee el valor de la propiedad **`pointerLockElement`**. Si la propiedad devuelve una referencia a un elemento, significa que el ratón se está bloqueando, pero si la propiedad devuelve el valor **`null`**, significa que ningún elemento está bloqueando al ratón en ese momento.

Hágalo usted mismo: actualice el documento en su archivo HTML con el código del Listado 13-3. Abra el documento en su navegador y active la consola para ver los mensajes. Haga clic en el elemento **<p>** para desactivar el ratón. Debería ver el mensaje "Ratón Bloqueado" en la consola. Pulse la tecla Escape. Debería ver el mensaje "Ratón Liberado" en la consola.

Lo básico: en la instrucción **`if`** dentro de la función **`controlarraton()`** usamos el valor de la propiedad **`pointerLockElement`** para establecer la condición. Esto es posible porque una condición siempre se considera verdadera a menos que el valor sea nulo, como 0, "", **`undefined`**, **`null`**, etc.

Como hemos mencionado, si ningún elemento tiene control sobre el ratón, la propiedad **`pointerLockElement`** devuelve el valor **`null`**, por lo que podemos usarla para decidir si bloquear o desbloquear el ratón dependiendo de la condición actual.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Pointer Lock</title>
  <script>
    var canvas;
    function iniciar() {
      var elemento = document.getElementById("canvas");
      canvas = elemento.getContext("2d");
      elemento.addEventListener("click", bloquearraton);
      elemento.addEventListener("mousemove", dibujar);
    }
    function dibujar(evento) {
      canvas.clearRect(0, 0, 500, 400);
      var posx = evento.clientX;
      var posy = evento.clientY;
      canvas.beginPath();
```

```
      canvas.moveTo(posx, posy - 20);
      canvas.lineTo(posx, posy + 20);
      canvas.moveTo(posx - 20, posy);
      canvas.lineTo(posx + 20, posy);
      canvas.moveTo(posx + 20, posy);
      canvas.arc(posx, posy, 20, 0, Math.PI * 2);
      canvas.stroke();
    }
    function bloquearraton(evento) {
      var elemento = evento.target;
      if (!document.pointerLockElement) {
        elemento.requestPointerLock();
      } else {
        document.exitPointerLock();
      }
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <canvas id="canvas" width="500" height="400"></canvas>
  </section>
</body>
</html>
```

Listado 13-4: *Bloqueando y desbloqueando el ratón*

En este ejemplo, creamos una pequeña aplicación que muestra un uso más realista de esta API. En este código, usamos la propiedad **pointerLockElement** para determinar si el lienzo tiene control sobre el ratón o no. Cada vez que el usuario hace clic en el elemento **<canvas>**, comprobamos esta condición y bloqueamos o desbloqueamos el ratón con **requestPointerLock()** o **exitPointerLock()** de acuerdo a las circunstancias.

Cuando el ratón se bloquea, el navegador asigna el control del ratón al elemento que ha hecho la solicitud. Todos los eventos del ratón, como **mousemove**, **click** o **mousewheel**, solo se desencadenan para este elemento, pero los eventos que están relacionados con el puntero del ratón, como **mouseover** o **mouseout**, ya no se desencadenan. Como resultado, para interactuar con el ratón y detectar sus movimientos, tenemos que responder al evento **mousemove**. El código del Listado 13-2 agrega un listener para el evento **mousemove** con el fin de dibujar la mira de un arma en el lienzo en la posición actual del ratón, determinada por las propiedades **clientX** y **clientY** (ver **MouseEvent** en el Capítulo 6).

Hágalo usted mismo: actualice el documento en su archivo HTML con el código del Listado 13-4. Abra el documento en su navegador y mueva el ratón al área ocupada por el elemento **<canvas>**. Debería ver la mira de un arma siguiendo al puntero. Haga clic para bloquear el ratón y fijar la mira en el lugar. Haga clic nuevamente para desbloquearlo.

En el ejemplo del Listado 13-4, la mira se mueve junto con el puntero del ratón, pero tan pronto como se bloquea el ratón, la mira queda congelada en la pantalla. Esto se debe a que cuando se bloquea el ratón, solo se actualizan los valores de las propiedades **movementX** y **movementY** (el resto de las propiedades del evento mantienen los valores almacenados antes de que el modo se

active). Estas propiedades no devuelven la posición exacta del ratón, sino la diferencia entre la posición actual y la anterior. Para poder trabajar con el ratón cuando está bloqueado, debemos calcular su posición a partir de los valores que devuelven estas propiedades.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Pointer Lock</title>
  <script>
    var canvas, posx, posy;
    function iniciar() {
      var elemento = document.getElementById("canvas");
      canvas = elemento.getContext("2d");
      elemento.addEventListener("click", bloquearraton);
      iniciomensaje();
    }
    function dibujar(evento) {
      canvas.clearRect(0, 0, 500, 400);
      var control1, control2;
      control1 = posx + event.movementX;
      control2 = posy + event.movementY;
      if(control1 > 0 && control1 < 500){
        posx = control1;
      }
      if(control2 > 0 && control2 < 400){
        posy = control2;
      }
      canvas.beginPath();
      canvas.moveTo(posx, posy - 20);
      canvas.lineTo(posx, posy + 20);
      canvas.moveTo(posx - 20, posy);
      canvas.lineTo(posx + 20, posy);
      canvas.moveTo(posx + 20, posy);
      canvas.arc(posx, posy, 20, 0, Math.PI * 2);
      canvas.stroke();
    }
    function bloquearraton(evento) {
      var elemento = evento.target;
      if (!document.pointerLockElement) {
        elemento.requestPointerLock();
        posx = evento.clientX;
        posy = evento.clientY;
        elemento.addEventListener("mousemove", dibujar);
      } else {
        document.exitPointerLock();
        elemento.removeEventListener("mousemove", dibujar);
        iniciomensaje();
      }
    }
    function iniciomensaje() {
      canvas.clearRect(0, 0, 500, 400);
      canvas.font = "bold 36px verdana, sans-serif";
      canvas.fillText("Clic para Jugar", 100, 180);
    }
```

```
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section>
    <canvas id="canvas" width="500" height="400"></canvas>
  </section>
</body>
</html>
```

Listado 13-5: *Calculando la posición del ratón con* `movementX` *y* `movementY`

En este código hemos introducido algunos cambios para incrementar el nivel de control y mostrar cómo podemos aprovechar esta API en ciertas fases de la aplicación. La función **`iniciar()`** configura el lienzo, agrega un listener para el evento **`click`** para bloquear el ratón, y llama a la función **`iniciomensaje()`** para mostrar un mensaje de bienvenida en la pantalla, pero no agrega un listener para el evento **`mousemove`**. Este listener se agrega en la función **`bloquearraton()`** después de que se bloquea el ratón y también se elimina en la misma función después de que el ratón se desbloquea. Siguiendo este procedimiento, el elemento **`<canvas>`** asume el control del ratón solo después de que se ejecute la primera etapa de la aplicación (donde le pedimos al usuario que haga clic en la pantalla). Si el usuario devuelve a esta primera etapa, el ratón se desbloquea de nuevo y el método **`removeEventListener()`** elimina el listener para el evento **`mousemove`**.

Los valores que devuelven las propiedades **`movementX`** y **`movementY`** reflejan el cambio en la posición. Antes de que el ratón se bloquee, capturamos sus coordenadas con las propiedades **`clientX`** y **`clientY`** para establecer la posición inicial de la mira. Normalmente, esto no es necesario, pero en una aplicación como esta, donde el ratón se bloquea y desbloquea constantemente, ayuda a producir una mejor transición de un estado a otro. Esta posición inicial se almacena en las variables **`posx`** y **`posy`** y solo se modifica mediante la cantidad de desplazamiento cuando no excede los límites establecidos por el elemento **`<canvas>`** (ver la instrucción **`if`** en la función **`dibujar()`**). A menos que solo estemos comprobando la dirección del ratón, este control es necesario para evitar almacenar valores que nuestra aplicación no pueda procesar.

Hágalo usted mismo: actualice el documento en su archivo HTML con el código del Listado 13-5. Abra el documento en su navegador. Haga clic en el lienzo para bloquear el ratón. Debería ver la mira moverse con el ratón, pero no el puntero del ratón. Haga clic nuevamente para volver a la pantalla inicial.

Capítulo 14

API Web Storage

14.1 Sistemas de almacenamiento

La API Web Storage nos permite almacenar datos en el disco duro del usuario y acceder a los mismos cuando el usuario vuelve a visitar nuestro sitio web. El sistema de almacenamiento provisto por esta API se puede usar en dos situaciones particulares: cuando la información tiene que estar disponible solo durante una sesión y cuando se tiene que preservar hasta que lo determina el código o el usuario. Con el propósito de clarificar esta situación para los desarrolladores, la API se ha dividido en dos partes llamadas *Session Storage* y *Local Storage*.

Session Storage—Este es un mecanismo de almacenamiento que mantiene los datos disponibles solo durante una sesión. A la información almacenada a través de este mecanismo se accede desde una ventana o pestaña, y se conserva hasta que se cierra la ventana.

Local Storage—Este mecanismo trabaja de forma similar al sistema de almacenamiento de una aplicación de escritorio. Los datos son se preservan de forma permanente y siempre están disponibles desde la aplicación que los ha creado.

Ambos mecanismos trabajan con una interfaz similar y comparten las mismas propiedades y métodos, y ambos dependen del origen, lo que significa que la información solo estará disponible para el sitio web que la ha generado. Cada sitio web tiene asignado un espacio de almacenamiento, y la información se preserva o elimina dependiendo del mecanismo aplicado.

14.2 Session Storage

El sistema Session Storage es el más sencillo de todos. Este sistema almacena los datos solo para una sesión, lo cual significa que los datos se eliminarán cuando el usuario cierre la ventana o pestaña. Las aplicaciones pueden usarlo como soporte o para almacenar información que se puede solicitar más adelante en el proceso pero que se vuelve irrelevante si el usuario abandona el sitio web.

El siguiente es el documento que vamos a utilizar para probar esta API. Como ambos sistemas trabajan con la misma interfaz, solo vamos a necesitar un documento y un formulario sencillo para probar los ejemplos de este capítulo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Web Storage</title>
  <link rel="stylesheet" href="almacenamiento.css">
  <script src="almacenamiento.js"></script>
</head>
```

```
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p><label>Clave: <input type="text" name="clave"
id="clave"></label></p>
      <p><label>Valor: <textarea name="texto"
id="texto"></textarea></label></p>
      <p><button type="button" id="grabar">Grabar</button></p>
    </form>
  </section>
  <section id="cajadatos">
    <p>Información no disponible</p>
  </section>
</body>
</html>
```

Listado 14-1: *Creando un documento para trabajar con la API Web Storage*

También necesitamos algunos estilos para diferenciar el formulario de la caja donde vamos a mostrar los datos.

```
#cajaformulario {
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos {
  float: left;
  width: 400px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
#clave, #texto {
  width: 200px;
}
#cajadatos > div {
  padding: 5px;
  border-bottom: 1px solid #999999;
}
```

Listado 14-2: *Diseñando la interfaz*

Hágalo usted mismo: cree un archivo HTML con el documento del Listado 14-1 y un archivo CSS llamado almacenamiento.css con los estilos del Listado 14-2. También necesitará un archivo JavaScript llamado almacenamiento.js para grabar y probar los códigos presentados a continuación.

Almacenando datos

Los datos se almacenan como ítems, que se componen de un par nombre/valor. Estos ítems son como variables, cada uno con un nombre y un valor, que se pueden crear, modificar o

eliminar. Los siguientes son los métodos con los que cuenta la API para crear y leer un ítem en el espacio de almacenamiento.

setItem(nombre, valor**)**—Este método crea y almacena un ítem con el nombre y el valor especificados por los atributos. Si ya existe un ítem con el mismo nombre, se actualizará con el nuevo valor, por lo que este método también se puede usar para modificar datos almacenados con anterioridad.

getItem(nombre**)**—Este método devuelve el valor del ítem con el nombre especificado por el atributo.

El objeto **`Window`** incluye dos propiedades para facilitar acceso a los sistemas de almacenamiento: **`sessionStorage`** y **`localStorage`**. Para almacenar y leer ítems, tenemos que ejecutar los métodos correspondientes desde estas propiedades, como en el siguiente ejemplo.

```
function iniciar() {
  var boton = document.getElementById("grabar");
  boton.addEventListener("click", nuevoitem);
}
function nuevoitem() {
  var clave = document.getElementById("clave").value;
  var valor = document.getElementById("texto").value;
  sessionStorage.setItem(clave, valor);
  mostrar(clave);
}
function mostrar(clave) {
  var cajadatos = document.getElementById("cajadatos");
  var valor = sessionStorage.getItem(clave);
  cajadatos.innerHTML = "<div>" + clave + " - " + valor + "</div>";
}
window.addEventListener("load", iniciar);
```

Listado 14-3: *Almacenando y leyendo datos*

En el código del Listado 14-3, la función **`nuevoitem()`** se ejecuta cada vez que el usuario hace clic en el botón Grabar del formulario. Esta función crea un ítem con la información insertada en los campos de entrada y luego llama a la función **`mostrar()`**. En esta función, el ítem se lee con el valor del atributo **`clave`** y el método **`getItem()`**, y luego el contenido del ítem se inserta en el elemento **`cajadatos`** para mostrarlo en la pantalla.

Hágalo usted mismo: copie el código del Listado 14-3 en su archivo almacenamiento.js y abra el documento del Listado 14-1 en su navegador. Inserte valores en los campos y pulse el botón para almacenarlos. Debería ver los valores del ítem que acaba de insertar dentro del elemento **`cajadatos`**.

Lo básico: los métodos y propiedades se pueden concatenar con notación de puntos. El intérprete procesa los componentes de la instrucción de izquierda a derecha. En el ejemplo del Listado 14-3, concatenamos el método **`getElementById()`** con la propiedad **`value`**. El intérprete primero ejecuta el método, obtiene una referencia al objeto **`Element`**, y luego lee el valor de la propiedad **`value`** de ese objeto. Este es simplemente un atajo, podríamos

haber almacenado la referencia al elemento en una variable y luego leer la propiedad desde esa variable en otra instrucción, como hemos hecho en ejemplos anteriores, pero de esta manera ahorramos algunas líneas de código. El resultado es el mismo, los valores que inserta el usuario en los campos se asignan a las variables **clave** y **valor**.

Además de estos métodos, la API también ofrece un atajo para crear y leer un ítem en el espacio de almacenamiento. Podemos usar el nombre del ítem como una propiedad de **sessionStorage** y acceder a su valor de esta manera. Como con cualquier otra propiedad, contamos con dos sintaxis: podemos encerrar la variable representando el nombre en corchetes (**sessionStorage[nombre] = valor**) o podemos usar notación de puntos (**sessionStorage.miitem = valor**).

```
function iniciar() {
  var boton = document.getElementById("grabar");
  boton.addEventListener("click", nuevoitem);
}
function nuevoitem() {
  var clave = document.getElementById("clave").value;
  var valor = document.getElementById("texto").value;
  sessionStorage[clave] = valor;
  mostrar(clave);
}
function mostrar(clave) {
  var cajadatos = document.getElementById("cajadatos");
  var valor = sessionStorage[clave];
  cajadatos.innerHTML = "<div>" + clave + " - " + valor + "</div>";
}
window.addEventListener("load", iniciar);
```

Listado 14-4: *Usando un atajo para trabajar con ítems*

Leyendo datos

Los ítems se almacenan en un array, por lo que también podemos acceder a los valores con un índice o un bucle. La API ofrece esta propiedad y este método con dicho propósito.

length—Esta propiedad devuelve el número de ítems acumulados en el espacio de almacenamiento de la aplicación.

key(índice**)**—Este método devuelve el nombre del ítem en el índice especificado por el atributo.

Los ejemplos anteriores solo leen el último ítem almacenado. Aprovechando el método **key()**, vamos a mejorar el código para listar todos los valores disponibles en el espacio de almacenamiento.

```
function iniciar() {
  var boton = document.getElementById("grabar");
  boton.addEventListener("click", nuevoitem);
  mostrar();
}
```

```
function nuevoitem() {
  var clave = document.getElementById("clave").value;
  var valor = document.getElementById("texto").value;
  sessionStorage.setItem(clave, valor);
  document.getElementById("clave").value = "";
  document.getElementById("texto").value = "";
  mostrar();
}
function mostrar() {
  var cajadatos = document.getElementById("cajadatos");
  cajadatos.innerHTML = "";
  for (var f = 0; f < sessionStorage.length; f++) {
    var clave = sessionStorage.key(f);
    var valor = sessionStorage.getItem(clave);
    cajadatos.innerHTML += "<div>" + clave + " - " + valor + "</div>";
  }
}
window.addEventListener("load", iniciar);
```

Listado 14-5: *Listando todos los ítems en el espacio de almacenamiento*

El propósito de este ejemplo es mostrar la lista completa de ítems en la pantalla. La función **`mostrar()`** se ha mejorado con la propiedad **`length`** y el método **`key()`**. Dentro de un bucle **`for`**, se llama al método **`key()`**para obtener el nombre de cada ítem. Por ejemplo, si el ítem en la posición 0 del espacio de almacenamiento se ha creado con el nombre "miitem", la instrucción **`sessionStorage.key(0)`** devolverá el valor de "miitem". Si llamamos a este método desde un bucle, podemos listar todos los ítems en la pantalla, cada uno con su correspondiente nombre y valor.

A la función **`mostrar()`** también se le llama al final de la función **`iniciar()`** para mostrar los ítems que ya se encuentran en el espacio de almacenamiento tan pronto como se ejecuta la aplicación.

Hágalo usted mismo: actualice su archivo almacenamiento.js con el código del Listado 14-5 y abra el documento del Listado 14-1 en su navegador. Inserte nuevos valores en el formulario y pulse el botón para almacenarlos. Debería ver una lista con todos los valores que ha insertado hasta el momento dentro del elemento **`cajadatos`**.

Lo básico: puede aprovechar la API Formularios estudiada en el Capítulo 7 para controlar la validez de los campos de entrada y no permitir la inserción de ítems no válidos o vacíos.

Eliminando datos

Los ítems se pueden crear, leer y, por supuesto, eliminar. La API incluye dos métodos para eliminar ítems del espacio de almacenamiento.

removeItem(nombre**)**—Este método elimina un ítem. El atributo **nombre** especifica el nombre del ítem a eliminar.

clear()—Este método elimina todos los ítems del espacio de almacenamiento.

El siguiente ejemplo incluye botones al lado de cada valor para eliminarlo.

```
function iniciar() {
  var boton = document.getElementById("grabar");
  boton.addEventListener("click", nuevoitem);
  mostrar();
}
function nuevoitem() {
  var clave = document.getElementById("clave").value;
  var valor = document.getElementById("texto").value;
  sessionStorage.setItem(clave, valor);
  document.getElementById("clave").value = "";
  document.getElementById("texto").value = "";
  mostrar();
}
function mostrar() {
  var cajadatos = document.getElementById("cajadatos");
  cajadatos.innerHTML = '<div><input type="button"
onclick="removerTodo()" value="Eliminar Todos"></div>';
  for (var f = 0; f < sessionStorage.length; f++) {
    var clave = sessionStorage.key(f);
    var valor = sessionStorage.getItem(clave);
    cajadatos.innerHTML += "<div>" + clave + " - " + valor + "<br>";
    cajadatos.innerHTML += '<input type="button"
onclick="removerItem(\'' + clave + '\')" value="Remover"></div>';
  }
}
function removerItem(clave) {
  if (confirm("Está seguro?")) {
    sessionStorage.removeItem(clave);
    mostrar();
  }
}
function removerTodo() {
  if (confirm("Está seguro?")) {
    sessionStorage.clear();
    mostrar();
  }
}
window.addEventListener("load", iniciar);
```

Listado 14-6: *Eliminando ítems en el espacio de almacenamiento*

Las funciones **iniciar()** y **nuevoitem()** del Listado 14-6 son las mismas que las del ejemplo anterior. Solo ha cambiado la función **mostrar()** para incorporar botones con el atributo de evento **onclick** con los que llamar a las funciones que eliminarán un ítem individual o todos juntos. El código crea un botón Eliminar para cada ítem de la lista y también un único botón en la parte superior para borrar el espacio de almacenamiento completo.

Las funciones **removerItem()** y **removerTodo()** son responsables de eliminar el ítem seleccionado o limpiar el espacio de almacenamiento, respectivamente. Cada función llama a la función **mostrar()** al final para actualizar la lista de ítems en la pantalla.

Hágalo usted mismo: con el código del Listado 14-6 podrá ver cómo se procesa la información con el sistema Session Storage. Abra el documento HTML del Listado 14-1 en su navegador, cree nuevos ítems y luego abra el mismo documento en otra ventana. La información será diferente en cada ventana. La primera ventana mantiene sus datos disponibles, pero el espacio de almacenamiento de la nueva ventana estará vacío. A diferencia de otros sistemas, Session Storage considera cada ventana como una instancia independiente de la aplicación y la información de la sesión no se comparte entre ellas.

14.3 Local Storage

Contar con un sistema fiable para almacenar datos durante una sesión puede ser útil en algunas circunstancias, pero cuando intentamos emular aplicaciones de escritorio en la Web, un sistema de almacenamiento temporario como este es generalmente insuficiente. Para mantener la información siempre disponible, la API Web Storage incluye el sistema Local Storage. Con Local Storage, podemos almacenar grandes cantidades de datos y dejar que el usuario decida si la información aún es útil y debemos conservarla o no.

Este sistema usa la misma interfaz que Session Storage; por lo tanto, cada método y propiedad estudiados en este capítulo están disponibles también en Local Storage. Solo necesitamos sustituir la propiedad **`sessionStorage`** por la propiedad **`localStorage`** para preparar los códigos.

```
function iniciar() {
  var boton = document.getElementById("grabar");
  boton.addEventListener("click", nuevoitem);
  mostrar();
}
function nuevoitem() {
  var clave = document.getElementById("clave").value;
  var valor = document.getElementById("texto").value;

  localStorage.setItem(clave, valor);
  mostrar();
  document.getElementById('clave').value = "";
  document.getElementById('texto').value = "";
}
function mostrar() {
  var cajadatos = document.getElementById("cajadatos");
  cajadatos.innerHTML = "";
  for (var f = 0; f < localStorage.length; f++) {
    var clave = localStorage.key(f);
    var valor = localStorage.getItem(clave);
    cajadatos.innerHTML += "<div>" + clave + " - " + valor + "</div>";
  }
}
window.addEventListener("load", iniciar);
```

Listado 14-7: *Usando Local Storage*

En el Listado 14-7, tomamos uno de los códigos anteriores y reemplazamos la propiedad **`sessionStorage`** por la propiedad **`localStorage`**. Ahora, cada ítem creado está disponible en cada ventana e incluso después de que el navegador se cierra por completo.

Hágalo usted mismo: usando el documento del Listado 14-1, pruebe el código del Listado 14-7. Este código crea un nuevo ítem con la información en el formulario y automáticamente lista todos los ítems disponibles en el espacio de almacenamiento reservado para la aplicación. Cierre el navegador y abra el documento nuevamente. Aún debería ver todos los ítems de la lista.

Evento storage

Debido a que la información almacenada por Local Storage está disponible en todas las ventanas donde se carga la aplicación, debemos resolver dos problemas: cómo se cominican estas ventanas entre sí y cómo actualizamos la información en la ventana que no está activa. La API incluye el siguiente evento para solucionar ambos problemas.

storage—La ventana desencadena este evento cada vez que ocurre un cambio en el espacio de almacenamiento. Se puede usar para informar a cada ventana abierta con la misma aplicación que se ha realizado un cambio en el espacio de almacenamiento y que se debe hacer algo al respecto.

Mantener la lista de ítems actualizada en nuestro ejemplo es fácil, solo tenemos que llamar a la función **`mostrar()`** cada vez que se desencadena el evento **`storage`**.

```
function iniciar() {
  var boton = document.getElementById("grabar");
  boton.addEventListener("click", nuevoitem);
  window.addEventListener("storage", mostrar);
  mostrar();
}
function nuevoitem() {
  var clave = document.getElementById("clave").value;
  var valor = document.getElementById("texto").value;
  localStorage.setItem(clave, valor);
  document.getElementById("clave").value = "";
  document.getElementById("texto").value = "";
  mostrar();
}
function mostrar() {
  var cajadatos = document.getElementById("cajadatos");
  cajadatos.innerHTML = "";
  for (var f = 0; f < localStorage.length; f++) {
    var clave = localStorage.key(f);
    var valor = localStorage.getItem(clave);
    cajadatos.innerHTML += "<div>" + clave + " - " + valor + "</div>";
  }
}
window.addEventListener("load", iniciar);
```

Listado 14-8: *Respondiendo al evento* `storage` *para mantener la lista de ítems actualizada*

En este ejemplo, la función **`mostrar()`** responde al evento **`storage`** y, por lo tanto, se ejecuta cada vez que se crea, actualiza o elimina un ítem. Ahora, si algo cambia en una ventana, se mostrará automáticamente en las otras ventanas que están ejecutando la misma aplicación.

Hágalo usted mismo: usando el documento del Listado 14-1, pruebe el código del Listado 14-8. El evento **storage** solo trabaja cuando la aplicación se almacena en un servidor o en un servidor local. Suba los archivos a su servidor y abra el documento en dos ventanas diferentes. Inserte o actualice un ítem en una ventana y abra la otra ventana para ver cómo el evento **storage** actualiza la información.

El evento **storage** envía a la función un objeto de tipo **StorageEvent** que contiene las siguientes propiedades para proveer información acerca de la modificación producida en el espacio de almacenamiento.

key—Esta propiedad devuelve el nombre del ítem afectado.

oldValue—Esta propiedad devuelve el valor anterior del ítem afectado.

newValue—Esta propiedad devuelve el nuevo valor asignado al ítem.

url—Esta propiedad devuelve la URL del documento que ha producido la modificación. El espacio de almacenamiento se reserva para todo el dominio, por lo que pueden acceder a él diferentes documentos.

storageArea—Esta propiedad devuelve un objeto que contiene todos los pares nombre/valor disponibles en el espacio de almacenamiento después de la modificación.

El siguiente ejemplo imprime en la consola los valores de estas propiedades para mostrar la información disponible.

```
function iniciar() {
  var boton = document.getElementById("grabar");
  boton.addEventListener("click", nuevoitem);
  addEventListener("storage", modificado);
  mostrar();
}
function nuevoitem() {
  var clave = document.getElementById("clave").value;
  var valor = document.getElementById("texto").value;
  localStorage.setItem(clave, valor);
  document.getElementById("clave").value = "";
  document.getElementById("texto").value = "";
  mostrar();
}
function modificado(evento) {
  console.log(evento.key);
  console.log(evento.oldValue);
  console.log(evento.newValue);
  console.log(evento.url);
  console.log(evento.storageArea);
  mostrar();
}
function mostrar() {
  var cajadatos = document.getElementById("cajadatos");
  cajadatos.innerHTML = "";
  for (var f = 0; f < localStorage.length; f++) {
    var clave = localStorage.key(f);
```

```
    var valor = localStorage.getItem(clave);
    cajadatos.innerHTML += "<div>" + clave + " - " + valor + "</div>";
  }
}
window.addEventListener("load", iniciar);
```

Listado 14-9: *Accediendo a las propiedades del evento*

En el Listado 14-9 agregamos una nueva función para responder al evento **`storage`**. Se llama a la función **`modificado()`** mediante el evento cuando el espacio de almacenamiento queda modificado por otra ventana. En esta función, imprimimos los valores de las propiedades uno por uno en la consola y al final llamamos a la función **`mostrar()`** para actualizar la información en la pantalla.

Capítulo 15
API IndexedDB

15.1 Datos estructurados

La API Web Storage que hemos estudiado en el capítulo anterior es útil para almacenar pequeñas cantidades de datos, pero cuando necesitamos trabajar con grandes cantidades de datos estructurados, debemos recurrir a un sistema de base de datos. Para este propósito, los navegadores incluyen la API IndexedDB.

La API IndexedDB es un sistema de base de datos capaz de almacenar información indexada en el disco duro del usuario. Se desarrolló como una API de bajo nivel con la intención de permitir una amplia gama de usos. Esto no solo la convirtió en una de las API más potentes, sino también en una de las más complejas. El objetivo fue el de facilitar la estructura más básica posible para permitir a los desarrolladores construir sistemas personalizados basados en la misma y crear interfaces de alto nivel para cada necesidad. En una API de bajo nivel como esta, tenemos que hacernos cargo de todo el proceso y controlar las condiciones de cada operación realizada. El resultado es una API a la que la mayoría de los desarrolladores les costará tiempo familiarizarse y probablemente la aplicarán a través de librerías de terceros.

La estructura de datos propuesta por la API IndexedDB es diferente de SQL u otros sistemas populares de bases de datos. La información se almacena en la base de datos como objetos (registros) dentro de lo que llamamos *Almacenes de objetos* (tablas). Los Almacenes de objetos no tienen una estructura predefinida, solo un nombre e índices para encontrar los objetos en su interior. Estos objetos tampoco tienen una estructura predefinida; pueden ser diferentes entre ellos y tan complejos como necesitemos. La única condición que deben cumplir los objetos es tener al menos una propiedad declarada como índice para poder encontrarlos dentro de los Almacenes de objetos.

Base de datos

La base de datos misma es sencilla. Debido a que cada base de datos se asocia a un ordenador y un sitio web o aplicación, no hay usuarios que tengamos que asignar o cualquier otra restricción de acceso que debamos considerar. Solo tenemos que especificar el nombre y la versión, y la base de datos estará lista.

El objeto `Window` incluye la propiedad `indexedDB` para acceder a la base de datos. Esta propiedad almacena un objeto con los siguientes métodos.

open(nombre, versión**)**—Este método crea o abre una base de datos con el nombre y las versión especificados por los atributos. La versión se puede ignorar cuando se crea la base, pero es obligatoria cuando queremos especificar una nueva versión para una base de datos existente. El método devuelve un objeto que desencadena dos eventos (`error` y `success`) para indicar error o éxito en la creación o apertura de la base de datos.

deleteDatabase(nombre**)**—Este método elimina la base de datos con el nombre especificado por el atributo.

La API ofrece la oportunidad de asignar una versión a la base de datos para poder actualizar su estructura. Cuando tenemos que actualizar la estructura de una base de datos en el servidor para agregar más tablas o índices, normalmente creamos una segunda base de datos con la nueva estructura y luego movemos los datos desde la base de datos antigua a la nueva. Este proceso requiere tener acceso completo al servidor e incluso la habilidad de apagar el servidor momentáneamente. Sin embargo, no podemos apagar el ordenador del usuario para repetir este proceso en el navegador. En consecuencia, se tiene que declarar un número de versión en el método `open()` para poder migrar la información de la versión antigua a la nueva. Para ayudarnos a trabajar con diferentes versiones de bases de datos, la API incluye el siguiente evento.

upgradeneeded—Este evento se desencadena cuando el método `open()` intenta abrir una base de datos no existente o cuando se ha especificado una nueva versión para la base de datos actual.

Objetos y almacenes de objetos

Lo que normalmente llamamos *registros* en una base de datos estándar se llaman *objetos* en la API IndexedDB. Estos objetos incluyen propiedades para almacenar e identificar valores. El número de propiedades y cómo se estructuran los objetos es irrelevante; solo tienen que incluir al menos una propiedad declarada como índice para que el Almacén de objetos pueda encontrarlos.

Del mismo modo, lo que llamamos *tablas* en una base de datos estándar se denomina *Almacenes de objetos* en la API IndexedDB, porque almacenan objetos con la información que queremos preservar. Los Almacenes de objetos tampoco tienen una estructura particular, solo se deben declarar el nombre y uno o más índices cuando se crean los objetos en su interior.

Almacén de Objetos

Objeto 1	Objeto 2
Id Nombre DVD Video	Id Nombre DVD
Objeto 3	**Objeto 4**
Id Nombre DVD Libro	Id Nombre Libro

Figura 15-1: *Objetos con diferentes propiedades almacenados en un almacén de objetos*

Como muestra la Figura 15-1, un Almacén de Objetos puede contener varios objetos con diferentes propiedades. En este ejemplo, algunos objetos tiene la propiedad `DVD`, otros tiene la propiedad `Libro`, etc. Cada uno tiene las suyas propias, pero todos deben incluir al menos un propiedad seleccionada como índice para que el Almacén de objetos pueda encontrarlos (en este ejemplo, la propiedad asignada con este propósito podría ser `Id`, porque todos los objetos la contienen).

Para trabajar con objetos y Almacenes de objetos, necesitamos crear el Almacén de objetos, declarar las propiedades que se usarán como índices y luego comenzar a almacenar los objetos en el mismo. Actualmente no tenemos que pensar en la estructura y el contenido de los objetos, solo tenemos que considerar los índices que vamos a necesitar para realizar búsquedas más adelante. Los siguientes son los métodos que facilita la API para gestionar Almacenes de objetos.

createObjectStore(nombre, objeto**)**—Este método crea un nuevo Almacén de objetos con el nombre y la configuración especificados por los atributos (el atributo **nombre** es obligatorio). El atributo **objeto** es un objeto que puede incluir las propiedades **keyPath** y **autoIncrement**. La propiedad **keyPath** declara un índice común para todos los objetos y la propiedad **autoIncrement** es una propiedad booleana que determina si el Almacén de objetos tendrá un generador de claves para organizar los objetos.

deleteObjectStore(nombre**)**—Este método elimina el Almacén de objetos con el nombre declarado por el atributo.

objectStore(nombre**)**—Este método abre el Almacén de objetos con el nombre declarado por el atributo.

Los métodos **createObjectStore()** y **deleteObjectStore()**, así como otros métodos responsables de la configuración de la base de datos, solo se pueden aplicar cuando se crea la base de datos o se actualiza a una nueva versión.

La API IndexedDB facilita los siguientes métodos para interactuar con el Almacén de objetos, leer y almacenar información.

add(objeto**)**—Este método recibe un par nombre/valor o un objeto que contiene varios pares nombre/valor y agrega un objeto al Almacén de objetos con esta información. Si ya existe un objeto con el mismo índice, este método devuelve un error.

put(objeto**)**—Este método es similar al método **add(),** excepto porque sobrescribe un objeto existente con el mismo índice. Es útil para actualizar un objeto que ya se ha almacenado en el Almacén de objetos.

get(nombre**)**—Este método recupera un objeto del Almacén de objetos. El atributo **nombre** es el valor del índice del objeto que queremos leer.

delete(nombre**)**—Este método elimina un objeto del Almacén de objetos. El atributo **nombre** es el valor del índice del objeto que queremos eliminar.

Índices

Como ya hemos mencionado, tenemos que declarar algunas de las propiedades de los objetos como índices para encontrarlos luego en el Almacén de objetos. Una manera sencilla de hacerlo es declarando la propiedad **keyPath** en el método **createObjectStore()**. La propiedad declarada como **keyPath** será el índice común compartido por todos los objetos almacenados en ese Almacén de objetos en concreto. Cuando declaramos el valor de **keyPath**, la propiedad declarada como índice debe estar presente en todos los objetos.

Además de **keyPath**, también podemos declarar los índices que queremos para un Almacén de objetos usando los siguientes métodos.

createIndex(nombre, propiedad, objeto**)**—Este método crea un índice para un Almacén de objetos. El atributo **nombre** es un nombre que identifica el índice, el atributo **propiedad** es la propiedad del objeto usada como índice y el atributo **objeto** es un objeto que puede incluir las propiedades **unique** o **multiEntry**. La propiedad **unique** indica si el valor del índice debe ser único o hay varios objetos que pueden compartir el mismo valor. La propiedad **multiEntry** determina cuántas claves se generarán para un objeto cuando el índice es un array.

deleteIndex(nombre**)**—Este método elimina un índice. Solo se puede llamar desde una transacción de tipo `versionchange`.

index(nombre**)**—Este método devuelve una referencia al índice con el nombre especificado por el atributo.

Transacciones

Un sistema de base de datos que funciona en un navegador debe considerar circunstancias únicas que no están presentes en otras plataformas. Por ejemplo, el navegador puede fallar, se puede cerrar abruptamente, el usuario puede detener el proceso o el usuario puede abandonar nuestro sitio web en medio de una operación. Existen muchas situaciones en las que trabajar directamente con la base de datos puede causar un mal funcionamiento o incluso corromper los datos. Con el fin de prevenir que esto suceda, se debe realizar cada acción a través de transacciones. La API IndexedDB ofrece el siguiente método para realizar una transacción.

transaction(almacén, tipo**)**—Este método realiza una transacción. El atributo **almacén** es el nombre del Almacén de objetos involucrado en la transacción y el atributo **tipo** es una cadena de caracteres que describe el tipo de transacción que queremos realizar.

Los siguientes son los tipos de transacciones disponibles para este método.

readonly—Esta es una transacción de solo lectura. No se permiten modificaciones.

readwrite—Esta es una transacción de lectura-escritura. Usando este tipo de transacción podemos leer y escribir en la base de datos. Se permiten modificaciones, pero no podemos agregar o eliminar Almacenes de objetos e índices.

versionchange—Este tipo de transacción se usa para actualizar la versión de la base de datos, así como para agregar o eliminar Almacenes de objetos e índices.

Las transacciones más comunes son `readwrite`. Sin embargo, para prevenir un uso inadecuado, se declara por defecto el tipo `readonly`, por lo que si solo necesitamos obtener información de la base de datos, solo tenemos que especificar el objetivo de la transacción (generalmente el nombre del Almacén de objetos) y el tipo de transacción se declara automáticamente.

Se facilitan los siguientes eventos para controlar la transacción.

complete—Este evento se desencadena cuando se completa la transacción.

abort—Este evento se desencadena cuando se anula la transacción.

error—Este evento se desencadena cuando ocurre un error durante la transacción.

15.2 Implementación

Es hora de crear nuestra primera base de datos y aplicar algunos de los métodos que hemos mencionado en este capítulo. En esta sección vamos a simular una aplicación para almacenar información sobre películas. Estos son los datos que vamos a utilizar en nuestros ejemplos.

id: tt0068646 **nombre:** El Padrino **fecha:** 1972
id: tt0086567 **nombre:** Juegos de Guerra **fecha:** 1983

id: tt0111161 **nombre:** Cadena Perpetua **fecha:** 1994
id: tt1285016 **nombre:** La Red Social **fecha:** 2010

IMPORTANTE: los nombres de estas propiedades (**`id`**, **`nombre`**, y **`fecha`**) se van a usar en los ejemplos del resto de este capítulo. La información se ha recogido del sitio web www.imdb.com, pero puede crear su propia lista o usar información al azar para probar los códigos.

Como siempre, necesitamos un documento HTML y algunos estilos CSS para definir las cajas del formulario y el espacio donde vamos a mostrar la información que devuelve la base de datos. El formulario incorpora campos para insertar información acerca de las películas, incluidos la clave, el título de la película y el año en el que se filmó.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API IndexedDB</title>
  <link rel="stylesheet" href="indexed.css">
  <script src="indexed.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <label for="clave">Clave: </label><br>
      <input type="text" name="clave" id="clave"><br>
      <label for="texto">Título: </label><br>
      <input type="text" name="texto" id="texto"><br>
      <label for="fecha">Año: </label><br>
      <input type="text" name="fecha" id="fecha"><br>
      <button type="button" id="grabar">Grabar</button>
    </form>
  </section>
  <section id="cajadatos">
    <p>Información no disponible</p>
  </section>
</body>
</html>
```

Listado 15-1: *Creando un documento para probar la API IndexedDB*

Los estilos CSS definen las cajas para el formulario y los datos.

```
#cajaformulario {
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos {
  float: left;
  width: 400px;
  margin-left: 20px;
  padding: 20px;
```

```
  border: 1px solid #999999;
}
#clave, #texto {
  width: 200px;
}
#cajadatos > div {
  padding: 5px;
  border-bottom: 1px solid #999999;
}
```

Listado 15-2: *Definiendo los estilos de las cajas*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 15-1, un archivo CSS llamado indexed.css con los estilos del Listado 15-2 y un archivo JavaScript llamado indexed.js para todos los código introducidos a continuación.

Abriendo la base de datos

El primer paso en el código JavaScript es abrir la base de datos. El método **`open()`** del objeto **`indexedDB`** abre la base de datos con el nombre especificado o crea una nueva si no existe.

```
var cajadatos, bd;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  var boton = document.getElementById("grabar");
  boton.addEventListener("click", agregarobjeto);
  var solicitud = indexedDB.open("basededatos");
  solicitud.addEventListener("error", mostrarerror);
  solicitud.addEventListener("success", comenzar);
  solicitud.addEventListener("upgradeneeded", crearbd);
}
```

Listado 15-3: *Abriendo la base de datos*

La función **`iniciar()`** del Listado 15-3 crea las referencias a los elementos del documento y abre la base de datos. El método **`open()`** intenta abrir una base de datos con el nombre "basededatos" y devuelve un objeto con el resultado. Debido a que la API IndexedDB es una API asíncrona, los eventos **`error`**, **`success`** y **`upgradeneeded`** se desencadenan desde este objeto para informar del resultado de la operación. Cuando se desencadenan los eventos **`error`** y **`success`**, se ejecutan las funciones **`mostrarerror()`** y **`comenzar()`** para controlar los errores o continuar con la definición de la base de datos.

```
function mostrarerror(evento) {
  alert("Error: " + evento.code + " " + evento.message);
}
function comenzar(evento) {
  bd = evento.target.result;
}
```

Listado 15-4: *Reportando errores y obteniendo una referencia a la base de datos*

Debido a que en esta aplicación no necesitamos procesar errores o hacer otra cosa que obtener una referencia a la base de datos, las funciones **mostrarerror()** y **comenzar()** son sencillas. La información del error se muestra usando las propiedades **code** y **message** del objeto **Event**, y se captura una referencia a la base de datos mediante la propiedad **result** y luego se almacena en la variable global **bd**. Esta variable se usará para acceder a la base de datos desde el resto del código.

Las funciones que responden a estos eventos reciben un objeto de tipo **IDBRequest**. Estos tipos de objetos incluyen propiedades con información acerca de la operación. Las siguientes son las que más se usan.

result—Esta propiedad devuelve un objeto con el resultado de la solicitud. Este puede ser un objeto que representa la base de datos o un objeto que representa el objeto obtenido del Almacén de objetos.

source—Esta propiedad devuelve un objeto con información acerca de la fuente de la solicitud.

transaction—Esta propiedad devuelve un objeto con información acerca de la transacción.

readyState—Esta propiedad devuelve la condición actual de la transacción. Los valores disponibles son **pending** y **done**.

error—Esta propiedad devuelve el error de la solicitud.

En nuestro ejemplo, vamos a usar solo la propiedad **result** para obtener una referencia a la base de datos y los objetos desde los Almacenes de objetos, como hemos hecho en el Listado 15-4.

Definiendo índices

En el caso de que se declare una nueva versión de la base de datos mediante el método **open()** o de que no exista la base de datos, el evento **upgradeneeded** se desencadena y se llama a la función **crearbd()** para responder al evento. En este momento, tenemos que pensar en la clase de objetos que necesitamos almacenar en la base de datos y cómo vamos a obtener esta información después desde los Almacenes de objetos.

```
function crearbd(evento) {
  var basededatos = evento.target.result;
  var almacen = basededatos.createObjectStore("peliculas", {keyPath:
"id"});
  almacen.createIndex("BuscarFecha", "fecha", {unique: false});
}
```

Listado 15-5: *Declarando Almacenes de objetos e índices*

Para nuestro ejemplo, solo necesitamos un Almacén de objetos (para almacenar las películas) y dos índices. El primer índice, **id**, se declara como el atributo **keyPath** del método **createObjectStore()** cuando se crea el Almacén de objetos, pero el segundo índice se asigna al Almacén de objetos usando el método **createIndex()**. Este índice se identifica con

el nombre **BuscarFecha** y se declara para la propiedad **fecha**. Vamos a usar este índice para ordenar las películas por año.

En esta función hemos tenido que obtener la referencia a la base de datos nuevamente desde la propiedad **result** porque el evento **success** aún no se ha desencadenado y la referencia a la base de datos todavía no se ha creado en nuestro código (el valor de la variable **bd** aún no se ha definido).

IMPORTANTE: si la estructura es incorrecta o más adelante queremos agregar contenido a la configuración de nuestra base de datos, tendremos que declarar una nueva versión y migrar los datos desde la anterior o modificar la estructura a través de una transacción **versionchange**.

Agregando objetos

Hasta aquí tenemos una base de datos con el nombre *basededatos* y un Almacén de objetos llamado *peliculas* con dos índices: **id** y **fecha** (llamado **BuscarFecha**). Es hora de comenzar a agregar objetos a este almacén.

```
function agregarobjeto() {
  var clave = document.getElementById("clave").value;
  var titulo = document.getElementById("texto").value;
  var fecha = document.getElementById("fecha").value;

  var transaccion = bd.transaction(["peliculas"], "readwrite");
  var almacen = transaccion.objectStore("peliculas");
  transaccion.addEventListener("complete", function() {
    mostrar(clave);
  });
  var solicitud = almacen.add({id: clave, nombre: titulo, fecha: fecha});

  document.getElementById("clave").value = "";
  document.getElementById("texto").value = "";
  document.getElementById("fecha").value = "";
}
```

Listado 15-6: *Agregando objetos a nuestro Almacén de objetos*

Al comienzo de la función **iniciar()**, agregamos un listener para el evento **click** al botón del formulario. La función **agregarobjeto()** del Listado 15-6 se ejecuta cuando se desencadena este evento. Esta función toma los valores insertados en el formulario (**clave**, **texto** y **fecha**) y genera una transacción para almacenar un nuevo objeto usando esta información.

La transacción se inicia llamando al método **transaction()** y especificando el Almacén de objetos involucrado en la transacción y su tipo. En este caso, el único Almacén disponible es peliculas y el tipo se declara como **readwrite**.

El siguiente paso es seleccionar el Almacén de objetos que vamos a utilizar. Debido a que la transacción puede iniciar para varios Almacenes de objetos, tenemos que declarar cuál corresponde a la operación que queremos realizar. Con el método **objectStore()** abrimos el Almacén de objetos y lo asignamos a la transacción con la instrucción **transaccion.objectStore("peliculas")**.

La mayoría de las transacciones van a ser consultas a la base de datos y, en esos casos, podemos responder al evento **success** de la solicitud para obtener los resultados (como veremos pronto), pero este evento se puede desencadenar antes de que se produzca un error en la transacción. Por ello, cuando agregamos o modificamos el contenido de la base de datos, es mejor responder al evento **complete** de la transacción. Es por esto que en nuestro ejemplo hemos asignado la función **mostrar()** a este evento. Como siempre, también existe un evento **error**, pero como la respuesta a este evento depende de los requerimientos de la aplicación, no vamos a considerar esta posibilidad en nuestro ejemplo.

Finalmente, es hora de agregar el objeto al Almacén de objetos. En este caso, usamos el método **add()** porque queremos crear nuevos objetos, pero podríamos haber usado el método **put()** si hubiésemos querido modificar o reemplazar objetos existentes. El método **add()** usa las propiedades **id**, **nombre** y **fecha,** y las variables **clave**, **titulo** y **fecha,** y crea el objeto usando estos valores como pares nombre/valor.

Leyendo objetos

Si el objeto se almacena correctamente, se desencadena el evento **complete** y se ejecuta la función **mostrar()**. En el código del Listado 15-6 esta función se ha llamado desde dentro de una función anónima para poder pasar la variable **clave**. Ahora vamos a usar este valor para leer el objeto que acabamos de almacenar.

```
function mostrar(clave) {
  var transaccion = bd.transaction(["peliculas"]);
  var almacen = transaccion.objectStore("peliculas");
  var solicitud = almacen.get(clave);
  solicitud.addEventListener("success", mostrarlista);
}
function mostrarlista(evento) {
  var resultado = evento.target.result;
  cajadatos.innerHTML = "<div>" + resultado.id + " - " +
resultado.nombre + " - " + resultado.fecha + "</div>";
}
```

Listado 15-7: *Leyendo el nuevo objeto*

Las funciones del Listado 15-7 generan una transacción **readonly** y usan el método **get()** para obtener un objeto con el nombre recibido (no tenemos que declarar el tipo de transacción porque el tipo **readonly** se declara por defecto).

El método **get()** devuelve el objeto almacenado con la propiedad **id = clave**. Si, por ejemplo, insertamos la película *El Padrino*, la variable **clave** tendrá el valor "tt0068646". Este valor lo recibe la función **mostrar()** y lo usa el método **get()** para obtener la película *El Padrino*. Por supuesto, este código se usa con propósitos ilustrativos porque solo devuelve la misma película que acabamos de almacenar.

Debido a que toda operación es asíncrona, necesitamos dos funciones para mostrar esta información. La función **mostrar()** genera la transacción y la función **mostrarlista()** muestra el valor de las propiedades en la pantalla en caso de éxito. Esta vez solo respondemos al evento **success** de la solicitud, pero también se desencadenará un evento **error** desde esta operación si algo sale mal.

La función **mostrarlista()** toma el objeto que devuelve la propiedad **result** y lo almacena en la variable **resultado**. El valor es el objeto obtenido del Almacén de objetos, por lo que para acceder a su contenido escribimos la variable que representa al objeto y el nombre de la propiedad, como en **resultado.id**. En este caso, la variable **resultado** representa al objeto e **id** es una de sus propiedades.

Como en todos los ejemplos anteriores, para finalizar la aplicación, debemos responder al evento **load** para ejecutar la función **iniciar()** después de que se cargue el documento.

```
window.addEventListener("load", iniciar);
```

Listado 15-8: *Iniciando la aplicación*

Hágalo usted mismo: copie todos los códigos JavaScript desde el Listado 15-3 al 15-8 dentro del archivo indexed.js y abra el documento HTML del Listado 15-1 en su navegador. Usando el formulario en la pantalla, inserte las películas incluidas en la lista inicial del capítulo. Cada vez que se inserta una nueva película, se muestra la misma información en la caja de la derecha.

15.3 Listando datos

El método **get()** implementado en el código del Listado 15-7 devuelve solo un objeto a la vez (la última película insertada por el usuario). Para generar una lista que incluya todos los objetos almacenados en un Almacén de objetos, tenemos que usar un cursor.

Cursores

EL cursor es la herramienta que facilita la API para leer y navegar a través de un grupo de objetos que devuelve la base de datos en una transacción. El cursor obtiene una lista de objetos desde el Almacén de objetos e inicializa un puntero que apunta a uno de los objetos de la lista a la vez.

La API incluye el método **openCursor()** para generar un cursor. Este método extrae información desde el Almacén de objetos seleccionado y devuelve un objeto **IDBCursor** que tiene sus propios métodos para gestionar el cursor.

continue(índice**)**—Este método mueve el puntero del cursor hacia adelante una posición y desencadena el evento **success** en el cursor para informar que la operación ha finalizado. El evento envía los valores del objeto obtenido a la función que responde al mismo. Cuando el puntero alcanza el final de la lista, el evento **success** también se desencadena, pero el objeto enviado a la función es un objeto vacío. El puntero se puede mover a una posición específica declarando un índice como atributo.

delete()—Este método elimina el objeto en la posición del cursor.

update(valor**)**—Este método es similar a **put()**, pero actualiza el valor del objeto en la posición actual del cursor.

El siguiente ejemplo ilustra cómo trabajar con un cursor.

```
var cajadatos, bd;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
```

```
  var boton = document.getElementById("grabar");
  boton.addEventListener("click", agregarobjeto);
  var solicitud = indexedDB.open("basededatos");
  solicitud.addEventListener("error", mostrarerror);
  solicitud.addEventListener("success", comenzar);
  solicitud.addEventListener("upgradeneeded", crearbd);
}
function mostrarerror(evento) {
  alert("Error: " + evento.code + " " + evento.message);
}
function comenzar(evento) {
  bd = evento.target.result;
  mostrar();
}
function crearbd(evento) {
  var basededatos = evento.target.result;
  var almacen = basededatos.createObjectStore("peliculas", {keyPath: "id"});
  almacen.createIndex("BuscarFecha", "fecha", {unique: false});
}
function agregarobjeto() {
  var clave = document.getElementById("clave").value;
  var titulo = document.getElementById("texto").value;
  var fecha = document.getElementById("fecha").value;
  var transaccion = bd.transaction(["peliculas"], "readwrite");
  var almacen = transaccion.objectStore("peliculas");
  transaccion.addEventListener("complete", mostrar);
  var solicitud = almacen.add({id: clave, nombre: titulo, fecha: fecha});
  document.getElementById("clave").value = "";
  document.getElementById("texto").value = "";
  document.getElementById("fecha").value = "";
}
function mostrar() {
  cajadatos.innerHTML = "";
  var transaccion = bd.transaction(["peliculas"]);
  var almacen = transaccion.objectStore("peliculas");
  var puntero = almacen.openCursor();
  puntero.addEventListener("success", mostrarlista);
}
function mostrarlista(evento) {
  var puntero = evento.target.result;
  if (puntero) {
    cajadatos.innerHTML += "<div>" + puntero.value.id + " - " +
puntero.value.nombre + " - " + puntero.value.fecha + "</div>";
    puntero.continue();
  }
}
window.addEventListener("load", iniciar);
```

Listado 15-9: *Listando objetos*

El Listado 15-9 incluye todo el código JavaScript necesario para este ejemplo. De todas las funciones utilizadas para inicializar y configurar la base de datos, solo **`comenzar()`** presenta un cambio. Ahora, la función **`mostrar()`** se llama al final de esta función para mostrar la lista de objetos dentro del Almacén de objetos en la pantalla tan pronto como se carga el

documento. Los cambios significativos en este código se encuentran en las funciones **`mostrar()`** y **`mostrarlista()`**, donde trabajamos con cursores por primera vez.

La lectura de información de la base de datos con un cursor es también una operación que debe realizarse a través de una transacción. En consecuencia, el primer paso en la función **`mostrar()`** es generar una transacción **`readonly`** para el Almacén de objetos peliculas. Este Almacén de objetos se selecciona para esta transacción y luego el cursor se abre con el método **`openCursor()`**. Si la operación se realiza correctamente, se devuelve un objeto con la información obtenida del Almacén de objetos, un evento **`success`** se desencadena desde este objeto y se ejecuta la función **`mostrarlista()`**. El objeto recibido por esta función incluye las siguientes propiedades para leer la información.

key—Esta propiedad devuelve el nombre del objeto en la posición actual del cursor.

value—Esta propiedad devuelve un objeto con los valores de las propiedades del objeto en la posición actual del cursor.

direction—Esta propiedad devuelve el orden en el que se leen objetos (ascendente o descendente).

count—Esta propiedad devuelve el número aproximado de objetos en el cursor.

En la función **`mostrarlista()`** del Listado 15-9 usamos una instrucción **`if`** para controlar el contenido del cursor. Si no se devuelve ningún objeto, o el puntero alcanza el final de la lista, el valor de la variable **`cursor`** será **`null`** y el bucle finalizará. Sin embargo, cuando el puntero apunta a un objeto válido, la información se muestra en pantalla y el puntero se mueve a la siguiente posición con **`continue()`**.

Es importante mencionar que no tenemos que usar un bucle en este caso porque el método **`continue()`** desencadena el evento **`success`** y toda la función se ejecuta otra vez hasta que el cursor devuelve **`null`**.

Hágalo usted mismo: el código del Listado 15-9 reemplaza todos los códigos JavaScript anteriores. Copie este código dentro del archivo indexed.js. Abra el documento del Listado 15-1 en su navegador y, si aún no lo ha hecho, inserte todas las películas incluidas en la lista inicial de este capítulo. Debería ver la lista completa de películas en la caja de la derecha en orden ascendente según el valor de la propiedad **`id`**.

Orden

Las películas de nuestro ejemplo se listan en orden ascendente y la propiedad que se usa para organizar los objetos es **`id`**. Esta propiedad es el valor de **`keyPath`** para el Almacén de objetos peliculas, pero no es un valor que será de interés para los usuarios. Considerando esta situación, en la función **`crearbd()`** del Listado 15-9 agregamos otro índice a nuestro almacén. El nombre de este índice adicional es **`BuscarFecha`** y la propiedad asignada al índice es **`fecha`**. Este índice nos permite ordenar las películas según el año en que se filmaron. Los siguientes son los cambios que necesitamos introducir en la función **`mostrar()`** con este propósito.

```
function mostrar() {
  cajadatos.innerHTML = "";
  var transaccion = bd.transaction(["peliculas"]);
```

```
  var almacen = transaccion.objectStore("peliculas");
  var indice = almacen.index("BuscarFecha");
  var puntero = indice.openCursor(null, "prev");
  puntero.addEventListener("success", mostrarlista);
}
```

Listado 15-10: Listando objetos por año en orden descendente

La función del Listado 15-10 reemplaza a la función **`mostrar()`** del Listado 15-9. Esta nueva función genera una transacción, luego asigna el índice **`BuscarFecha`** al Almacén de objetos usado en la transacción y finalmente llama al método **`openCursor()`** para obtener la lista de objetos que contienen la propiedad correspondiente a ese índice (en este caso, **`fecha`**).

Existen dos atributos que podemos incluir en el método **`openCursor()`** para seleccionar y ordenar la información que devuelve el cursor. El primer atributo especifica el rango con el que seleccionar los objetos y el segundo atributo es una de las siguientes constantes.

next—Los objetos se devuelven en orden ascendente (por defecto).

nextunique—Los objetos se devuelven en orden ascendente y los objetos duplicados se ignoran (solo se devuelve el primer objeto si se encuentra un nombre duplicado).

prev—Los objetos se devuelven en orden ascendente.

prevunique—Los objetos se devuelven en orden descendente y los objetos duplicados se ignoran (solo se devuelve el primer objeto si se encuentra un nombre duplicado).

En el método **`openCursor()`** dentro de la función **`mostrar()`** del Listado 15-10 declaramos el atributo para el rango como **`null`** y el orden de los objetos como descendiente. Estos valores devuelven todos los objetos en orden descendiente. Vamos a estudiar cómo construir un rango al final de este capítulo.

Hágalo usted mismo: tome el código del Listado 15-9 y reemplace la función **`mostrar()`** con la nueva función del Listado 15-10. Esta nueva función lista las películas en la pantalla por año y en orden descendente.

15.4 Eliminando datos

Ya hemos aprendido a agregar, leer y listar datos. Es hora de ver cómo eliminar objetos del Almacén de objetos. Como hemos mencionado antes, el método **`delete()`** que facilita la API recibe un valor y elimina el objeto con el nombre correspondiente a ese valor. El código para nuestro ejemplo es sencillo; solo tenemos que modificar la función **`mostrarlista()`** para crear botones por cada objeto listado en la pantalla y generar una transacción **`readwrite`** para poder eliminarlos cuando se eliminan los botones.

```
function mostrarlista(evento) {
  var puntero = evento.target.result;
  if (puntero) {
    cajadatos.innerHTML += "<div>" + puntero.value.id + " - " +
puntero.value.nombre + " - ";
```

```
    cajadatos.innerHTML += puntero.value.fecha + ' <input type="button"
onclick="removerobjeto(\'' + puntero.value.id + '\')"
value="Remover"></div>';
    puntero.continue();
  }
}
function removerobjeto(clave) {
  if (confirm("Está seguro?")) {
    var transaccion = bd.transaction(["peliculas"], "readwrite");
    var almacen = transaccion.objectStore("peliculas");
    transaccion.addEventListener("complete", mostrar);
    var solicitud = almacen.delete(clave);
  }
}
```

Listado 15-11: *Eliminando objetos*

Los botones agregados a cada objeto en la función **mostrarlista()** del Listado 15-11 contienen un atributo de evento. Cada vez que el usuario hace clic en uno de estos botones, la función **removerobjeto()** se ejecuta con el valor de la propiedad **id** del objeto como atributo. Esta función genera una transacción **readwrite** y luego, usando el valor del atributo **clave**, procede a eliminar el objeto correspondiente del Almacén de objetos peliculas. Al final, si la transacción se realiza correctamente, se desencadena el evento **complete** y se ejecuta la función **mostrar()** para actualizar la lista de películas en pantalla.

Hágalo usted mismo: reemplace la función **mostrarlista()** en el código del Listado 15-9 y agregue la función **removerobjeto()** del Listado 15-11. Abra el documento del Listado 15-1 en su navegador para probar la aplicación. Debería ver la lista de películas, pero ahora, cada línea incluye un botón para eliminar la película del Almacén de objetos.

15.5 Buscando datos

La operación más importante que se realiza en una base de datos es probablemente la búsqueda de datos. El propósito de este tipo de sistemas es el de organizar la información para que sea fácil de encontrar. Como hemos visto antes en este capítulo, el método **get()** es útil para obtener un objeto a la vez cuando conocemos su nombre, pero una operación de búsqueda es más complicada.

Con el fin de obtener una lista específica de objetos desde el Almacén de objetos, tenemos que especificar un rango como el primer atributo del método **openCursor()**. La API facilita el objeto **IDBKeyRange** con varios métodos para declarar un rango y filtrar los objetos devueltos.

only(valor**)**—Este método devuelve un rango donde solo se devuelven los objetos con el valor igual al atributo **valor**. Por ejemplo, si buscamos películas por año usando **only("1972")**, la película *El Padrino* será la única que devuelve nuestra lista.

bound(bajo, alto, bajoAbierto, altoAbierto**)**—Este método devuelve un rango cerrado con valores de inicio y finalización. El atributo **bajo** especifica el valor de inicio del rango, el atributo **alto** especifica el valor final del rango y los atributos **bajoAbierto** y **altoAbierto** son valores booleanos que declaran si se ignorarán los objetos que coinciden

con los valores de los atributos **bajo** y **alto**. Por ejemplo, **`bound("1972", "2010", false, true)`** devuelve la lista de películas filmadas desde 1972 a 2010, pero no incluye las realizadas en el año 2010.

lowerBound(valor, abierto**)**—Este método crea un rango abierto que comienza por el valor especificado por el atributo **valor** y va hasta el final de la lista. Por ejemplo, **`lowerBound("1983", true)`** devuelve todas las películas hechas después de 1983, sin incluir las que se filmaron ese año.

upperBound(valor, abierto**)**—Este método crea un rango abierto, pero a diferencia del método **`lowerBound()`** los objetos que devuelve son los que se encuentran desde el inicio de la lista hasta el valor especificado por el atributo **valor**. Por ejemplo, **`upperBound("1983", false)`** devuelve las películas hechas antes de 1983, incluidas las realizadas ese año.

Para este ejemplo vamos a necesitar un nuevo documento que incluye un formulario para buscar películas.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API IndexedDB</title>
  <link rel="stylesheet" href="indexed.css">
  <script src="indexed.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <label for="fecha">Find Movie by Year: </label><br>
      <input type="text" name="fecha" id="fecha"><br>
      <button type="button" id="buscar">Buscar</button>
    </form>
  </section>
  <section id="cajadatos">
    <p>Información no disponible</p>
  </section>
</body>
</html>
```

Listado 15-12: *Creando un documento para buscar películas*

Este nuevo documento incluye un campo de entrada para permitirnos introducir el año de la película que queremos encontrar. El siguiente código crea un rango desde ese valor.

```
var cajadatos, bd;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  var boton = document.getElementById("buscar");
  boton.addEventListener("click", buscarobjetos);
  var solicitud = indexedDB.open("basededatos");
  solicitud.addEventListener("error", mostrarerror);
  solicitud.addEventListener("success", comenzar);
```

```
  solicitud.addEventListener("upgradeneeded", crearbd);
}
function mostrarerror(evento) {
  alert("Error: " + evento.code + " " + evento.message);
}
function comenzar(evento) {
  bd = evento.target.result;
}
function crearbd(evento) {
  var basededatos = evento.target.result;
  var almacen = basededatos.createObjectStore("peliculas", {keyPath:
"id"});
  almacen.createIndex("BuscarFecha", "fecha", {unique: false});
}
function buscarobjetos() {
  cajadatos.innerHTML = "";
  var buscar = document.getElementById("fecha").value;
  var transaccion = bd.transaction(["peliculas"]);
  var almacen = transaccion.objectStore("peliculas");
  var indice = almacen.index("BuscarFecha");
  var rango = IDBKeyRange.only(buscar);
  var puntero = indice.openCursor(rango);
  puntero.addEventListener("success", mostrarlista);
}
function mostrarlista(evento) {
  var puntero = evento.target.result;
  if (puntero) {
    cajadatos.innerHTML += "<div>" + puntero.value.id + " - " +
puntero.value.nombre + " - " + puntero.value.fecha + "</div>";
    puntero.continue();
  }
}
window.addEventListener("load", iniciar);
```

Listado 15-13: *Buscando películas*

La función **buscarobjetos()** del Listado 15-13 genera una transacción **readonly** para el Almacén de objetos peliculas, selecciona el índice **BuscarFecha** para usar la propiedad **fecha** como índice y crea un rango que solo incluye los objetos con un valor igual al valor de la variable **buscar** (el año insertado en el formulario). Este rango se pasa como el atributo del método **openCursor()**. Después de una operación realizada correctamente, la función **mostrarlista()** imprime la lista de películas que coinciden con el año seleccionado en pantalla.

Hágalo usted mismo: el método **only()** devuelve solo las películas que coinciden con el valor de la variable **buscar**. Para probar otros métodos, puede introducir valores específicos para el resto de los atributos (por ejemplo, **bound(buscar, "2012", false, true)**.

Capítulo 16
API File

16.1 Archivos

La API File se creó para facilitar el acceso a archivos locales desde una aplicación web. Permite a nuestras aplicaciones trabajar con los archivos del usuario. Con esta API, podemos leer archivos, mostrar sus contenidos al usuario, procesarlos e incluso cortarlos en partes para un procesamiento avanzado o para enviarlos a un servidor.

Cargando archivos

Es peligroso trabajar con archivos locales desde una aplicación web. Los navegadores tienen que considerar medidas de seguridad antes de siquiera contemplar la posibilidad de permitir a las aplicaciones acceder a los archivos del usuario. Considerando estas restricciones, la API File solo admite dos formas de cargar un archivo: el elemento **<input>** y la operación de arrastrar y soltar. En los ejemplos de este capítulo, vamos a usar el elemento **<input>** y estudiaremos cómo trabajar con la API Drag and Drop (arrastrar y soltar) en el Capítulo 17. El siguiente documento ilustra cómo se implementa esta clase de campos de entrada.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API File</title>
  <link rel="stylesheet" href="file.css">
  <script src="file.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <label for="archivos">Archivo: </label>
      <input type="file" name="archivos" id="archivos">
    </form>
  </section>
  <section id="cajadatos">
    <p>Seleccione un archivo</p>
  </section>
</body>
</html>
```

Listado 16-1: *Creando un documento para cargar archivos*

Los estilos CSS para este documento definen dos cajas en la pantalla para separar el formulario del área donde vamos a mostrar la información obtenida desde los archivos.

```
#cajaformulario {
  float: left;
```

```
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos {
  float: left;
  width: 500px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
```

Listado 16-2: *Definiendo los estilos del formulario y el elemento* `cajadatos`

Leyendo archivos

Para leer y procesar el contenido de un archivo, la API define un objeto llamado **`FileReader`**. El siguiente es el constructor que facilita la API para crear este objeto.

FileReader()—Este constructor devuelve un objeto **`FileReader`**. El objeto se debe crear antes de leer los archivos. El proceso es asíncrono y el resultado se ofrece a través de eventos.

El objeto **`FileReader`** incluye los siguientes métodos para obtener el contenido de un archivo.

readAsText(archivo, codificar**)**—Este método procesa el contenido del archivo como texto. El contenido se devuelve como texto en el formato UTF-8 a menos que se especifique el atributo **codificar**. El método intentará interpretar cada byte o secuencia de bytes como caracteres de texto.

readAsBinaryString(archivo**)**—Este método lee el contenido del archivo como una sucesión de números enteros en un rango de 0 a 255. El método nos asegura que los datos se van a leer sin intentar interpretarlos. Es útil para procesar contenido binario como imágenes o vídeos.

readAsDataURL(archivo**)**—Este método genera un valor de tipo data:url codificado en base64 que representa los datos del archivo.

readAsArrayBuffer(archivo**)**—Este método genera datos en formato ArrayBuffer y representa los datos que contiene el archivo.

El siguiente código muestra cómo leer un archivo seleccionado por el usuario.

```
var cajadatos;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  var archivos = document.getElementById("archivos");
  archivos.addEventListener("change", procesar);
}
function procesar(evento) {
  var archivos = evento.target.files;
  var archivo = archivos[0];
```

```
  var lector = new FileReader();
  lector.addEventListener("load", mostrar);
  lector.readAsText(archivo);
}
function mostrar(evento) {
  var resultado = evento.target.result;
  cajadatos.innerHTML = resultado;
}
window.addEventListener("load", iniciar);
```

Listado 16-3: *Leyendo un archivo de texto*

El campo de entrada del documento del Listado 16-1 permite al usuario seleccionar un archivo para su procesamiento. Para detectar la selección, en la función **iniciar()** del Listado 16-3 hemos agregado un listener para el evento **change** del elemento **<input>** y se ha declarado la función **procesar()** para responder a este evento.

La propiedad **files** enviada por el elemento **<input>** es un array que contiene objetos de tipo **File** que representan todos los archivos seleccionados por el usuario. Como no hemos incluido el atributo **multiple** en el elemento **<input>**, el usuario no puede seleccionar múltiples archivos, por lo que el primer elemento del array es el único disponible en este caso. Al comienzo de la función **procesar()**, leemos este valor desde el array **files** y lo almacenamos en la variable **archivo** para procesarlo (**var archivo = archivos[0]**).

Lo primero que tenemos que hacer para procesar el archivo es usar el constructor **FileReader()** para obtener el objeto **FileReader**. En la función **procesar()** del Listado 16-3, llamamos a este objeto **lector**. A continuación, debemos agregar un listener para el evento **load** para detectar cuándo se ha cargado el archivo y está ya listo para ser procesado. Finalmente, el método **readAsText()** lee el archivo y procesa su contenido como texto.

Cuando el método **readAsText()** finaliza la lectura del archivo, se desencadena el evento **load** y se llama a la función **mostrar()**. Esta función lee el contenido del archivo desde la propiedad **result** del objeto **lector** y lo muestra en pantalla.

Este código, por supuesto, espera archivos de texto, pero el método **readAsText()** puede recibir cualquier clase de archivos y los interpreta como texto, incluidos los archivos de contenido binario, como imágenes. En consecuencia, se muestran en pantalla un montón de caracteres extraños cuando se selecciona un archivo binario.

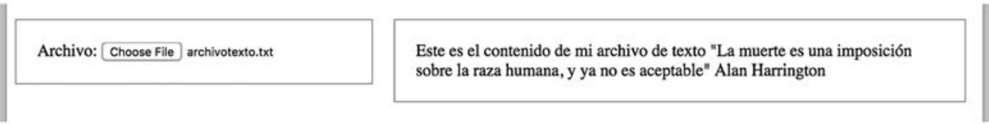

Figura 16-1: *Archivo de texto cargado desde el ordenador del usuario*

Hágalo usted mismo: cree archivos con los códigos de los Listados 16-1, 16-2 y 16-3. Los nombres de los archivos CSS y JavaScript se han declarado en el documento HTML como file.css y file.js, respectivamente. Abra el documento en su navegador y use el formulario para seleccionar un archivo desde su disco duro. Pruebe tanto archivos de texto como imágenes para ver cómo se muestra en pantalla el contenido de estos archivos.

Propiedades

En una aplicación real es necesario incluir información, como el nombre del archivo, su tamaño o su tipo, para informar al usuario acerca de los archivos que se están procesando o incluso para filtrar la entrada del usuario. El objeto **File** creado por el elemento **<input>** para representar cada archivo incluye algunas propiedades con este propósito.

name—Esta propiedad devuelve el nombre completo del archivo (nombre y extensión).

size—Esta propiedad devuelve el tamaño del archivo en bytes.

type—Esta propiedad devuelve el tipo MIME del archivo.

Si leemos y mostramos los valores de estas propiedades, podemos ayudar al usuario a identificar los archivos que ha cargado la aplicación.

```
var cajadatos;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  var archivos = document.getElementById("archivos");
  archivos.addEventListener("change", procesar);
}
function procesar(evento) {
  var archivos = evento.target.files;
  cajadatos.innerHTML = "";
  var archivo = archivos[0];
  if (!archivo.type.match(/image.*/i)) {
    alert("Insertar una imagen");
  } else {
    cajadatos.innerHTML += "Nombre: " + archivo.name + "<br>";
    cajadatos.innerHTML += "Tamaño: " + archivo.size + " bytes<br>";
    var lector = new FileReader();
    lector.addEventListener("load", mostrar);
    lector.readAsDataURL(archivo);
  }
}
function mostrar(evento) {
  var resultado = evento.target.result;
  cajadatos.innerHTML += '<img src="' + resultado + '">';
}
window.addEventListener("load", iniciar);
```

Listado 16-4: *Cargando imágenes*

El ejemplo del Listado 16-4 es similar al anterior, pero esta vez usamos el método **readAsDataURL()** para leer el archivo. Este método devuelve el contenido en formato data:url que se puede usar luego como fuente de un elemento **<img>** para mostrar la imagen seleccionada en la pantalla (ver Capítulo 11, Listado 11-29).

Cuando queremos procesar un tipo específico de archivo, el primer paso es leer la propiedad **type** del archivo. En la función **procesar()** del Listado 16-4 controlamos el valor de esta propiedad aplicando un método llamado **match()**, el cual compara un valor con una expresión regular y devuelve un array con los valores que coinciden con la expresión o el valor

null si no se encuentra ninguna coincidencia. Si el archivo no es una imagen, el método **alert()** muestra un mensaje de error. En caso contrario, se muestran en pantalla el nombre y tamaño de la imagen, y se abre el archivo.

A pesar del uso del método **readAsDataURL()**, el proceso para abrir el archivo es exactamente el mismo. Se crea el objeto **FileReader**, se agrega un listener al evento **load** y se carga el archivo. Una vez que el proceso finaliza, la función **mostrar()** usa el contenido de la propiedad **result** como fuente de un elemento **<img>** y la imagen se muestra en la pantalla.

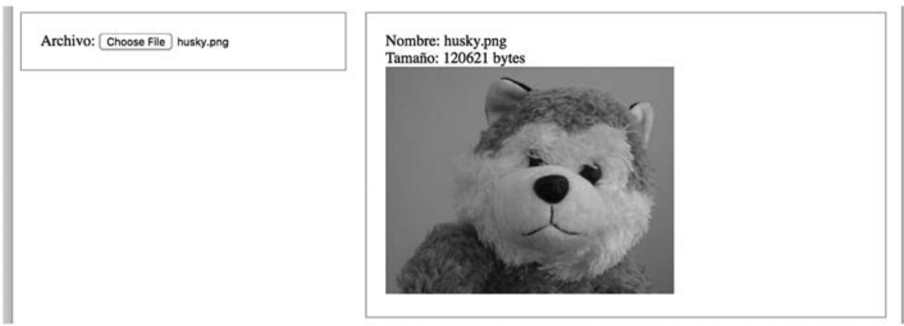

Figura 16-2: *Información acerca de una imagen en el ordenador del usuario*

Lo básico: como ilustra el ejemplo del Listado 16-4, las expresiones regulares y el método **match()** se pueden usar para filtrar información. Este método busca una coincidencia entre la expresión regular en paréntesis y la cadena de caracteres, y devuelve un array con todas las cadenas de caracteres que coinciden con la expresión o el valor **null** si no se encuentra ninguna. El tipo MIME para imágenes tiene una sintaxis **image/jpeg** (para imágenes JPG) o **image/gif** (para imágenes GIF), por lo que la expresión **/image.*/i** permite que solo se lean imágenes, independientemente de su formato. Para obtener más información sobre las expresiones regulares, el método **match()** y los tipos MIME, visite nuestro sitio web y siga los enlaces de este capítulo.

Blobs

Además del formato data:url, la API trabaja con otro tipo de formato llamado *blob*. Un blob es un objeto que contiene datos sin procesar. Se creó con la intención de superar las limitaciones que tenía JavaScript para trabajar con datos binarios. Normalmente, un objeto **Blob** se genera desde un archivo, pero esto no siempre es así. Los blobs son una buena alternativa para trabajar con datos sin tener que cargar el archivo entero en memoria, y ofrecen la posibilidad de procesar información binaria en trozos más pequeños.

Un blob tiene múltiples propósitos, pero está enfocado a ofrecer una mejor manera de procesar archivos extensos o gran cantidad de datos sin procesar. La API ofrece el siguiente método para generar objetos **Blob** desde otro blob o un archivo.

slice(comienzo, extensión, tipo**)**—Este método devuelve un objeto **Blob** generado a partir de otro blob o un archivo. El primer atributo indica el punto de comienzo, el segundo atributo indica la extensión y el tercer atributo especifica el tipo de datos (opcional).

El siguiente ejemplo corta un trozo de un archivo y muestra el resultado en pantalla.

```
var cajadatos;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  var archivos = document.getElementById("archivos");
  archivos.addEventListener("change", procesar);
}
function procesar(evento) {
  cajadatos.innerHTML = "";
  var archivos = evento.target.files;
  var archivo = archivos[0];
  var lector = new FileReader();
  lector.addEventListener("load", function(evento) {
    mostrar(evento, archivo)
  });
  var blob = archivo.slice(0,1000);
  lector.readAsBinaryString(blob);
}
function mostrar(evento, archivo){
  var resultado = evento.target.result;
  cajadatos.innerHTML = "Nombre: " + archivo.name + "<br>";
  cajadatos.innerHTML += "Tipo: " + archivo.type + "<br>";
  cajadatos.innerHTML += "Tamaño: " + archivo.size + " bytes<br>";
  cajadatos.innerHTML += "Tamaño del Blob: " + resultado.length + "
bytes<br>";
  cajadatos.innerHTML += "Blob: " + resultado;
}
window.addEventListener("load", iniciar);
```

Listado 16-5: *Trabajando con blobs y el método* `slice()`

En el código del Listado 16-5 hacemos lo mismo que hemos hecho hasta ahora, pero esta vez, en lugar de leer el archivo completo, creamos un blob con el método **`slice()`**. El blob tiene una extensión de 1000 bytes y comienza por el byte en la posición 0 del archivo. Si el tamaño del archivo es menor que 1000 bytes, el blob será tan largo como el archivo (desde la posición de inicio al EOF, o fin del archivo). Para mostrar la información obtenida por este proceso, respondemos al evento **`load`** con una función anónima. Esta función llama a la función **`mostrar()`** con una referencia al objeto **`archivo`** como su atributo. Esta referencia la recibe la función **`mostrar()`** y los valores de las propiedades de **`archivo`** se muestran en la pantalla.

Hágalo usted mismo: actualice su archivo file.js con el código del Listado 16-5 y abra el documento del Listado 16-1 en su navegador. Seleccione un archivo. Debería ver en la pantalla la información acerca del archivo junto con sus primeros 1000 caracteres.

Un blob no es apropiado como fuente de un elemento. Para convertirlo en una fuente para elementos como **`<img>`**, **`<video>`** o **`<audio>`**, tenemos que transformarlo en una URL. Los navegadores incluyen un objeto llamado **`URL`** diseñado para crear y procesar objetos **`URL`** que contienen todos los datos relacionados con una URL. Estos objetos incluyen dos métodos para trabajar con blobs, uno para crear un objeto **`URL`** desde un blob y otro para eliminarlo.

createObjectURL(datos**)**—Este método devuelve un objeto **`URL`** que podemos usar para referenciar los datos. El atributo **datos** puede ser un archivo, un blob o una transmisión de medios.

revokeObjectURL(URL**)**—Este método elimina una URL creada por el método **`createObjectURL()`**. Es útil para evitar el uso de viejas URL por códigos externos o por accidente desde nuestra propia aplicación.

JavaScript incluye un constructor de objetos **`Blob`** llamado **`Blob()`** que podemos usar para generar un blob desde otros tipos de datos o unir trozos de datos para formar un blob, pero cuando trabajamos con archivos, esto no es necesario. Los objetos **`File`** son blobs, por lo que podemos procesar estos objetos directamente como lo hacemos con blobs. Por ejemplo, podemos obtener un objeto **`File`** desde el archivo seleccionado por el usuario, convertirlo en una URL con el método **`createObjectURL()`**, y asignar esa URL a un elemento de medios para mostrar la imagen o el vídeo en la pantalla.

```
var cajadatos;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  var archivos = document.getElementById("archivos");
  archivos.addEventListener("change", procesar);
}
function procesar(evento) {
  cajadatos.innerHTML = "";
  var archivos = evento.target.files;
  var archivo = archivos[0];
  var lector = new FileReader();
  lector.addEventListener("load", function(evento) {
    mostrar(evento, archivo)
  });
  lector.readAsBinaryString(archivo);
}
function mostrar(evento, archivo){
  var url = URL.createObjectURL(archivo);
  var imagen = document.createElement("img");
  imagen.src = url;
  cajadatos.appendChild(imagen);
}
window.addEventListener("load", iniciar);
```

Listado 16-6: *Asignando un blob a un elemento* `<img>`

El ejemplo del Listado 16-6 asume que el usuario ha seleccionado un archivo de imagen. El proceso para leer el archivo es el mismo que hemos usado antes, pero ahora la función **`mostrar()`** convierte el objeto **`File`** en una URL y la asigna a un nuevo elemento **`<img>`** que se agrega al contenido del elemento **`cajadatos`** para mostrar la imagen en la pantalla.

Este es un ejemplo sencillo que convierte un archivo en un blob y luego obtiene la URL para mostrar el contenido del archivo en pantalla, sin procesar el archivo o su contenido, pero muestra las posibilidades que ofrecen los blobs. Con blobs, podemos crear aplicaciones para procesar imágenes, o un bucle que genere varios blobs a partir de un mismo archivo y luego subirlos a un servidor, entre otras.

Figura 16-3: *Imagen creada desde un blob*

Eventos

El tiempo que se tarda en cargar un archivo en memoria depende de su tamaño. Para archivos pequeños, el proceso parece una operación instantánea, pero los archivos grandes pueden tardar varios segundos en cargarse. Además del evento `load`, la API incluye una lista de eventos para ofrecer información en cada instancia del proceso.

loadstart—Este evento se desencadena mediante el objeto `FileReader` cuando comienza a leer un archivo.

progress—Este evento se desencadena periódicamente mientras se está leyendo el archivo o blob.

abort—Este evento se desencadena cuando se anula el proceso.

error—Este evento se desencadena cuando se produce un error en la lectura.

loadend—Este evento es similar a `load` pero se desencadena en caso de éxito o no.

El evento `progress` envía un objeto de tipo `ProgressEvent` a la función que responde al mismo. El objeto incluye tres propiedades para retornar información acerca del proceso que está siendo controlado por el evento.

lengthComputable—Esta es una propiedad booleana que devuelve `true` cuando el progreso se puede calcular o `false` en caso contrario. Lo usamos para asegurarnos de que los valores de las otras propiedades son válidos.

loaded—Esta propiedad devuelve el número total de bytes que ya se han descargado o subido.

total—Esta propiedad devuelve el tamaño total en bytes de los datos a descargar o subir.

El siguiente ejemplo crea una aplicación que carga un archivo y muestra el estado de la operación con una barra de progreso.

```
var cajadatos;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  var archivos = document.getElementById("archivos");
  archivos.addEventListener("change", procesar);
}
```

```
function procesar(evento) {
  cajadatos.innerHTML = "";
  var archivos = evento.target.files;
  var archivo = archivos[0];
  var lector = new FileReader();
  lector.addEventListener("loadstart", comenzar);
  lector.addEventListener("progress", estado);
  lector.addEventListener("loadend", function() {
    mostrar(archivo);
  });
  lector.readAsBinaryString(archivo);
}
function comenzar(evento) {
  cajadatos.innerHTML = '<progress value="0" max="100">0%</progress>';
}
function estado(evento) {
  var porcentaje = parseInt(evento.loaded / evento.total * 100);
  cajadatos.innerHTML = '<progress value="' + porcentaje + '"
max="100">' + porcentaje + '%</progress>';
}
function mostrar(archivo) {
  cajadatos.innerHTML = "Nombre: " + archivo.name + "<br>";
  cajadatos.innerHTML += "Tipo: " + archivo.type + "<br>";
  cajadatos.innerHTML += "Tamaño: " + archivo.size + " bytes<br>";
}
window.addEventListener("load", iniciar);
```

Listado 16-7: *Usando eventos para controlar el proceso*

En este ejemplo, hemos agregado tres listeners al objeto **FileReader** para controlar el proceso de lectura. Las funciones para responder a estos eventos son **comenzar()**, **estado()**, y **mostrar()**. La función **comenzar()** inicia la barra de progreso con el valor 0 % y la muestra en la pantalla cuando se desencadena el evento **loadstart**. Por otro lado, la función **estado()** recrea la barra de progreso cada vez que se desencadena el evento **progress**. El porcentaje se calcula desde el valor de las propiedades **loaded** y **total** recibidas desde el evento. Al final, la función **mostrar()** muestra la información del archivo que se ha procesado.

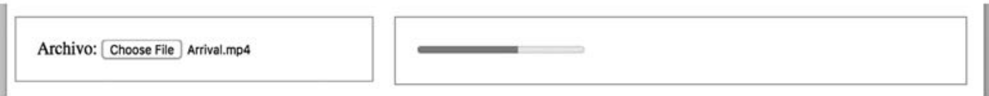

Figura 16-4: *Barra para mostrar el progreso mientras se descarga un archivo*

Hágalo usted mismo: intente cargar un archivo grande como un vídeo para poder ver la barra de progreso usando el documento del Listado 16-1 y el código JavaScript del Listado 16-7. Si el navegador no reconoce el elemento **<progress>**, en su lugar se muestra el contenido de este elemento.

Capítulo 17

API Drag and Drop

17.1 Arrastrar y soltar

La API Drag and Drop se incluyó para permitir a los usuarios arrastrar y soltar elementos y contenidos en la Web. La API define propiedades y métodos para controlar el proceso, pero el aspecto más importante es un grupo de siete eventos que se desencadenan en cada paso del proceso. Algunos de estos eventos se desencadenan desde la fuente (el elemento que se está arrastrando) y otros lo hacen mediante el destino (el elemento en el cual se suelta la fuente). Por ejemplo, cuando el usuario realiza una operación de arrastrar y soltar, la fuente desencadena los siguientes tres eventos.

dragstart—Este evento se desencadena en el momento en el que comienza la operación de arrastre. Los datos asociados con el elemento que se está arrastrando se almacenan en el sistema en este momento.

drag—Este evento es similar al evento `mousemove`, excepto porque se desencadena durante una operación de arrastre a través del elemento que se está arrastrando.

dragend—Este evento se desencadena cuando finaliza la operación de arrastre, independientemente de si se ha realizado correctamente o no.

Los siguientes son los eventos desencadenados por el elemento destino durante la misma operación.

dragenter—Este evento se desencadena cuando el ratón entra en el área de un posible elemento destino durante una operación de arrastre.

dragover—Este evento es similar al evento `mouseover`, excepto porque se desencadena durante una operación de arrastre a través de los posibles elementos destino.

drop—Este evento se desencadena cuando se suelta el elemento.

dragleave—Este evento se desencadena cuando el ratón abandona el área de un elemento durante una operación de arrastre. Se usa junto con `dragenter` para permitir que el usuario identifique el elemento destino.

En una operación de arrastrar y soltar necesitamos almacenar la información que se compartirá entre los elementos. Para este propósito, la API ofrece el objeto `DataTransfer`. Este objeto incluye varios métodos y propiedades con los que declarar y leer los datos. Los siguientes son los más usados.

types—Esta propiedad devuelve un array que contiene los tipos de datos declarados para los datos que se están transfiriendo.

files—Esta propiedad devuelve un array de objetos `File` que contienen información acerca de los archivos que se están arrastrando (los archivos se pueden arrastrar al navegador desde otras aplicaciones).

dropEffect—Esta propiedad devuelve el tipo de operación actualmente seleccionada. También se puede usar para cambiar la operación seleccionada. Los valores disponibles son **none**, **copy**, **link**, y **move**.

effectAllowed—Esta propiedad devuelve o declara los tipos de operaciones permitidas. Los valores disponibles son **none**, **copy**, **copyLink**, **copyMove**, **link**, **linkMove**, **move**, **all** y **uninitialized**.

setData(tipo, datos**)**—Este método se utiliza para declarar los datos que se enviarán y su tipo. El método acepta tipos MIME comunes (como **text/plain**, **text/html** o **text/uri-list**), tipos especiales (como **URL** o **Text**) e incluso tipos personalizados. Se debe realizar una llamada a este método por cada tipo de datos que queremos enviar en la misma operación.

getData(tipo**)**—Este método devuelve los datos del tipo especificado por el atributo.

clearData(tipo**)**—Este método elimina los datos del tipo especificado por el atributo.

setDragImage(elemento, x, y**)**—Este método se usa para personalizar la imagen en miniatura del elemento que se está arrastrando y establecer su posición respecto al puntero del ratón.

Antes de trabajar con esta API hay un aspecto importante que debemos considerar. Los navegadores realizan acciones por defecto durante una operación de arrastrar y soltar. Para obtener los resultados que queremos, debemos prevenir el comportamiento por defecto con el método **preventDefault()** y personalizar la respuesta. Para algunos eventos, como **dragenter**, **dragover** y **drop**, esta prevención es necesaria incluso cuando se ha especificado una acción personalizada. El siguiente ejemplo demuestra cómo implementar estos métodos y eventos.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Arrastrar y Soltar</title>
  <link rel="stylesheet" href="dragdrop.css">
  <script src="dragdrop.js"></script>
</head>
<body>
  <section id="deposito">
    <p>Arrastre y suelte la imagen aquí</p>
  </section>
  <section id="cajaimagenes">
    <img id="imagen" src="monstruo.gif">
  </section>
</body>
</html>
```

Listado 17-1: *Creando un documento para experimentar con la API Drag and Drop*

El documento del Listado 17-1 incluye un elemento **<section>** identificado como **deposito** que vamos a utilizar como destino, y una imagen que se usará como fuente.

También incluye dos archivos para los estilos CSS y el código JavaScript que controlará la operación. Los siguientes son los estilos para las cajas.

```
#deposito {
  float: left;
  width: 500px;
  height: 300px;
  margin: 10px;
  border: 1px solid #999999;
}
#cajaimagenes {
  float: left;
  width: 320px;
  margin: 10px;
  border: 1px solid #999999;
}
#cajaimagenes > img {
  float: left;
  padding: 5px;
}
```

Listado 17-2: *Definiendo los estilos para las cajas*

Las reglas del Listado 17-2 diseñan las cajas que ayudan al usuario a identificar la fuente y la caja donde depositarla. A continuación presentamos el código JavaScript que controla la operación.

```
var fuente, deposito;
function iniciar() {
  fuente = document.getElementById("imagen");
  fuente.addEventListener("dragstart", arrastrado);
  deposito = document.getElementById("deposito");
  deposito.addEventListener("dragenter", function(evento) {
    evento.preventDefault();
  });
  deposito.addEventListener("dragover", function(evento) {
    evento.preventDefault();
  });
  deposito.addEventListener("drop", soltado);
}
function arrastrado(evento) {
  var codigo = '<img src="' + fuente.src + '">';
  evento.dataTransfer.setData("Text", codigo);
}
function soltado(evento) {
  evento.preventDefault();
  deposito.innerHTML = evento.dataTransfer.getData("Text");
}
window.addEventListener("load", iniciar);
```

Listado 17-3: *Procesando una operación de arrastrar y soltar*

La aplicación debe responder a los eventos **dragstart** y **drop** para enviar y recibir los datos que queremos compartir entre los elementos, pero también tiene que responder a los

eventos **dragenter** y **dragover**. Esto se debe a que la acción de soltar normalmente no se permite en la mayoría de los elementos del documento por defecto. Por lo tanto, para habilitar esta operación para nuestra caja, debemos prevenir el comportamiento por defecto con el método **preventDefault()** cuando se desencadenan estos eventos.

El código para controlar el proceso es simple. Cuando el usuario comienza a arrastrar el elemento, se desencadena el evento **dragstart** y se llama a la función **arrastrado()**. En esta función obtenemos el valor del atributo **src** del elemento que se está arrastrando y declaramos los datos a transferir usando el método **setData()** del objeto **DataTransfer**. Desde el otro lado, cuando se suelta un elemento sobre la caja, se desencadena el evento **drop** y se llama a la función **soltado()**. Esta función modifica el contenido de la caja con la información obtenida por el método **getData()**. Los navegadores también realizan acciones por defecto cuando ocurre este evento (por ejemplo, abrir un enlace o actualizar la ventana con la imagen que se ha soltado), por lo que tenemos que prevenir este comportamiento usando el método **preventDefault()**, como ya hemos hecho con los otros eventos.

En la función **arrastrado()** del Listado 17-3 hemos creado un código HTML con el valor del atributo **src** del elemento que ha desencadenado el evento **dragstart** y lo enviamos como datos con el método **setData()**. Como estamos enviando texto, declaramos los datos con el tipo **Text**. El tipo de datos también se debe especificar cuando leemos los datos con el método **getData()**. Esto se debe a que se pueden enviar diferentes tipos de datos al mismo tiempo mediante el mismo elemento. Por ejemplo, una imagen podría enviar la imagen misma, la URL y una descripción. En casos como este, tenemos que enviar la información usando varios métodos **setData()** con diferentes tipos de valores y luego recuperarlos con métodos **getData()** especificando los mismos tipos.

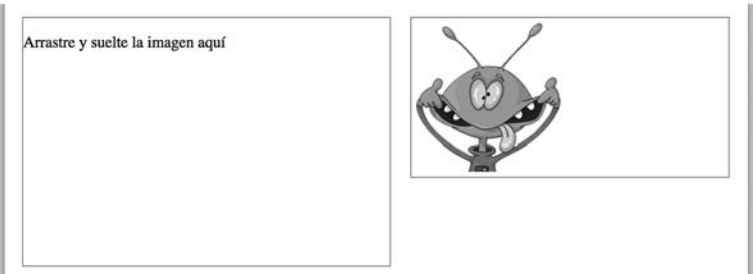

Figura 17-1: *Arrastrar y soltar*

Hágalo usted mismo: cree un archivo HTML con el documento del Listado 17-1, un archivo CSS llamado dragdrop.css con los estilos del Listado 17-2, y un archivo JavaScript llamado dragdrop.js con el código del Listado 17-3. Descargue la imagen monstruo.gif desde nuestro sitio web y muévalo al directorio de su proyecto. Abra el documento en su navegador y arrastre la imagen a la caja de la izquierda.

IMPORTANTE: la mayoría de los elementos, como las imágenes, se pueden arrastrar por defecto, pero en el caso de que queramos arrastrar y soltar un elemento como **<div>**, por ejemplo, la API facilita un atributo llamado **draggable**. Solo tenemos que agregar este atributo al elemento con el valor **true** para permitir que el usuario lo arrastre (por ejemplo, **<div draggable="true">**).

Hasta el momento solo hemos usado el evento **dragenter** para cancelar el comportamiento por defecto del navegador, y tampoco hemos aprovechado los eventos **dragleave** y **dragend**. El siguiente ejemplo implementa estos eventos para ofrecer una respuesta al usuario que le ayude a mover elementos de un lugar a otro.

```
var fuente, deposito;
function iniciar() {
  fuente = document.getElementById("imagen");
  fuente.addEventListener("dragstart", arrastrar);
  fuente.addEventListener("dragend", finalizar);
  deposito = document.getElementById("deposito");
  deposito.addEventListener("dragenter", entrar);
  deposito.addEventListener("dragleave", salir);
  deposito.addEventListener("dragover", function(evento) {
    evento.preventDefault();
  });
  deposito.addEventListener("drop", soltar);
}
function entrar(evento) {
  evento.preventDefault();
  deposito.style.background = "rgba(0, 150, 0, .2)";
}
function salir(evento) {
  evento.preventDefault();
  deposito.style.background = "#FFFFFF";
}
function finalizar(evento) {
  elemento = evento.target;
  elemento.style.visibility = "hidden";
}
function arrastrar(evento) {
  var codigo = '<img src="' + fuente.src + '">';
  evento.dataTransfer.setData("Text", codigo);
}
function soltar(evento) {
  evento.preventDefault();
  deposito.style.background = "#FFFFFF";
  deposito.innerHTML = evento.dataTransfer.getData("Text");
}
window.addEventListener("load", iniciar);
```

Listado 17-4: *Ofreciendo una respuesta al usuario*

El código JavaScript del Listado 17-4 reemplaza el código del Listado 17-3. En este ejemplo, agregamos tres funciones: **entrar()**, **salir()**, y **finalizar()**. Estas funciones responden a los eventos **dragenter**, **dragleave**, y **dragend**, respectivamente. Sus propósitos son el de ofrecer una ayuda visual. Las funciones **entrar()** y **salir()** cambian el color de fondo de la caja cada vez que el ratón arrastra algo, y entra o sale de la zona ocupada por el elemento, mientras que la función **finalizar()** modifica la propiedad **visibility** del elemento fuente para ocultarlo. En consecuencia, cada vez que el ratón arrastra algo y entra dentro del área de la caja de la izquierda, la caja se vuelve verde, y cuando se suelta el elemento, la imagen que se ha arrastrado, se oculta. Estos cambios visuales no afectan al proceso pero guían al usuario durante la operación.

Una vez más, para prevenir las acciones por defecto, tenemos que usar el método **`preventDefault()`** en cada función, incluso cuando se ha declarado una acción personalizada.

Hágalo usted mismo: reemplace el código del archivo JavaScript por el código del Listado 17-4, abra el documento del Listado 17-1 en su navegador y arrastre la imagen a la caja de la izquierda.

IMPORTANTE: el código del Listado 17-4 usa el evento **`dragend`** para ocultar la imagen original cuando finaliza la operación. Este evento se desencadena mediante la fuente cuando finaliza una operación de arrastrar, tanto si se realiza correctamente como si no. En nuestro ejemplo, la imagen se oculta en ambos casos. Debería crear los controles apropiados para proceder solo en caso de éxito. Veremos algunos ejemplos que ilustran cómo hacerlo.

Validación

No existe ningún método para detectar si la fuente es válida o no. No podemos confiar en la información que devuelve el método **`getData()`** porque incluso cuando solo obtenemos los datos del tipo que especificamos, otras fuentes podrían usar el mismo tipo y facilitar datos que no esperamos. El objeto **`DataTransfer`** incluye una propiedad llamada **`types`** que devuelve un array con una lista de los tipos que declara el evento **`dragstart`**, pero tampoco nos sirve con propósitos de validación. Por esta razón, las técnicas para seleccionar y validar los datos transferidos por una operación de arrastrar y soltar varían, y el procedimiento puede ser tan sencillo o complicado como lo requiera nuestra aplicación. El siguiente ejemplo valida la fuente leyendo el atributo **`id`** del elemento.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Arrastrar y Soltar</title>
  <link rel="stylesheet" href="dragdrop.css">
  <script src="dragdrop.js"></script>
</head>
<body>
  <section id="deposito">
    <p>Arrastre y suelte las imágenes aquí</p>
  </section>
  <section id="cajafotos">
    <img id="imagen1" src="monstruo1.gif">
    <img id="imagen2" src="monstruo2.gif">
    <img id="imagen3" src="monstruo3.gif">
    <img id="imagen4" src="monstruo4.gif">
  </section>
</body>
</html>
```

Listado 17-5: *Diseñando un documento para arrastrar y soltar varias fuentes*

El siguiente código lee el atributo **`id`** del elemento para seleccionar qué imagen se puede arrastrar y cuál no.

```
var deposito;
function iniciar() {
  var imagenes = document.querySelectorAll("#cajaimagenes > img");
  for (var i = 0; i < imagenes.length; i++) {
    imagenes[i].addEventListener("dragstart", arrastrar);
  }
  deposito = document.getElementById("deposito");
  deposito.addEventListener("dragenter", function(evento) {
    evento.preventDefault();
  });
  deposito.addEventListener("dragover", function(evento) {
    evento.preventDefault();
  });
  deposito.addEventListener("drop", soltar);
}
function arrastrar(evento) {
  elemento = evento.target;
  evento.dataTransfer.setData("Text", elemento.id);
}
function soltar(evento) {
  evento.preventDefault();
  var id = evento.dataTransfer.getData("Text");
  if (id != "imagen4") {
    var url = document.getElementById(id).src;
    deposito.innerHTML = '<img src="' + url + '">';
  } else {
    deposito.innerHTML = "No Admitido";
  }
}
window.addEventListener("load", iniciar);
```

Listado 17-6: *Filtrando las imágenes con el atributo* `id`

El código del Listado 17-6 no introduce muchos cambios con respecto a ejemplos anteriores. Usamos el método **`querySelectorAll()`** para agregar un listener para el evento **`dragstart`** a cada imagen dentro del elemento **`cajafotos`**, enviamos el valor del atributo **`id`** con **`setData()`** cada vez que se arrastra una imagen y controlamos este valor en la función **`soltar()`** para evitar que el usuario suelte la imagen con el atributo **`id`** igual a **`"imagen4"`** (el mensaje "No Admitido" se muestra en la caja cuando el usuario intenta soltar esta imagen en concreto).

Figura 17-2: *La imagen número 4 no se admite*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 17-5, un archivo CSS llamado dragdrop.css con los estilos del Listado 17-2 y un archivo JavaScript llamado dragdrop.js con el código del Listado 17-6. Descargue las imágenes monstruo1.gif, monstruo2.gif, monstruo3.gif y monstruo4.gif desde nuestro sitio web y muévalas al directorio de su proyecto. Abra el documento en su navegador y arrastre las imágenes a la caja de la izquierda. Debería ver el mensaje "No Admitido" dentro de la caja cuando intente soltar la imagen con el identificador "imagen4".

Lo básico: este es un filtro sencillo. Puede mejorarlo comprobando que la imagen que se recibe se encuentra dentro del elemento **`cajafotos`**, por ejemplo, o usar propiedades del objeto **`DataTransfer`** (como **`types`** o **`files`**), pero es siempre un proceso personalizado. En otras palabras, tiene que hacerse cargo usted mismo y adaptar el proceso a las características de su aplicación.

Imagen miniatura

Los navegadores generan una imagen miniatura con una imagen de la fuente y la arrastran junto con el puntero del ratón. La posición de esta imagen se establece de acuerdo a la posición del puntero del ratón cuando se inicia la operación de arrastrar. Si usamos el método **`setDragImage()`**, podemos especificar una nueva posición y también personalizar la imagen. El siguiente ejemplo muestra la importancia de este método al usar un elemento **`<canvas>`** como la caja destino.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Arrastrar y Soltar</title>
  <link rel="stylesheet" href="dragdrop.css">
  <script src="dragdrop.js"></script>
</head>
<body>
  <section id="deposito">
    <canvas id="canvas" width="500" height="300"></canvas>
  </section>
  <section id="cajaimagenes">
    <img id="imagen1" src="monstruo1.gif">
    <img id="imagen2" src="monstruo2.gif">
    <img id="imagen3" src="monstruo3.gif">
    <img id="imagen4" src="monstruo4.gif">
  </section>
</body>
</html>
```

Listado 17-7: *Usando un elemento* `<canvas>` *para recibir las imágenes*

El código JavaScript para esta aplicación debe implementar técnicas convencionales para procesar la imagen y establecer su posición cuando se suelta.

```
var deposito, canvas;
function iniciar() {
  var imagenes = document.querySelectorAll("#cajaimagenes > img");
  for (var i = 0; i < imagenes.length; i++) {
    imagenes[i].addEventListener("dragstart", arrastrar);
    imagenes[i].addEventListener("dragend", finalizar);
  }
  deposito = document.getElementById("canvas");
  canvas = deposito.getContext("2d");
  deposito.addEventListener("dragenter", function(evento) {
    evento.preventDefault();
  });
  deposito.addEventListener("dragover", function(evento) {
    evento.preventDefault();
  });
  deposito.addEventListener("drop", soltar);
}
function finalizar(evento) {
  elemento = evento.target;
  elemento.style.visibility = "hidden";
}
function arrastrar(evento) {
  elemento = evento.target;
  evento.dataTransfer.setData("Text", elemento.id);
  evento.dataTransfer.setDragImage(elemento, 0, 0);
}
function soltar(evento) {
  evento.preventDefault();
  var id = evento.dataTransfer.getData("Text");
  var elemento = document.getElementById(id);
  var posx = evento.pageX - deposito.offsetLeft;
  var posy = evento.pageY - deposito.offsetTop;
  canvas.drawImage(elemento, posx, posy);
}
window.addEventListener("load", iniciar);
```

Listado 17-8: *Configurando la imagen miniatura*

Con este ejemplo nos acercamos a una aplicación de la vida real. El código del Listado 17-8 realiza tres tareas: personaliza la imagen que se arrastrará con el método **`setDragImage()`**, dibuja la imagen en el lienzo usando el método **`drawImage()`** y oculta la imagen fuente cuando el proceso finaliza.

Para personalizar la imagen, usamos el mismo elemento que se está arrastrando. No cambiamos este aspecto, pero declaramos su posición como **`0,0`**. Esto significa que ahora sabemos dónde se ubica la imagen en relación al ratón. Usando esta información, calculamos la posición exacta en la que se ha soltado en el lienzo y dibujamos la imagen en esa ubicación precisa, ayudando al usuario a encontrar el lugar correcto en el que soltarla.

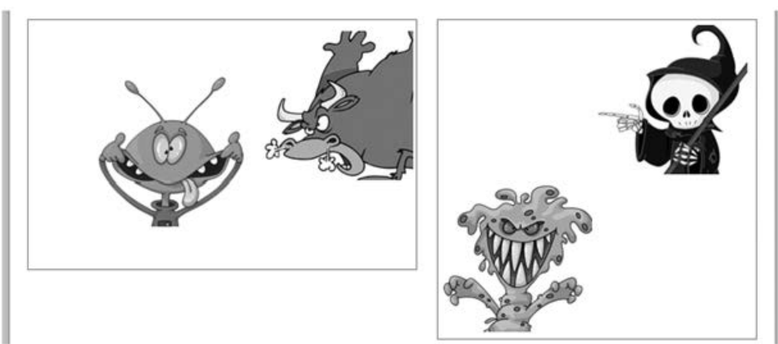

Figura 17-3: *Imágenes arrastradas dentro de un lienzo*

Archivos

La API no solo está disponible dentro del documento, sino también integrada en el sistema, lo que permite a los usuarios arrastrar elementos desde el navegador hacia otras aplicaciones y viceversa. Debido a esta característica, podemos arrastrar archivos desde aplicaciones externas a nuestro documento.

Como hemos visto antes, hay una propiedad que se ha incluido con este propósito dentro del objeto **`DataTransfer`** llamada **`files`**, la cual devuelve un array de objetos **`File`** que representa los archivos que se han arrastrado. Podemos usar esta información para construir códigos complejos que ayuden al usuario a trabajar con archivos o subirlos a un servidor.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Arrastrar y Soltar</title>
  <link rel="stylesheet" href="dragdrop.css">
  <script src="dragdrop.js"></script>
</head>
<body>
  <section id="deposito">
    <p>Arrastre y suelte los archivos aquí</p>
  </section>
</body>
</html>
```

Listado 17-9: *Creando un documento sencillo para arrastrar archivos*

El documento del Listado 17-9 incluye nuevamente un elemento **`<section>`** para crear una caja en la que soltar los archivos. Los archivos se arrastrarán y soltarán dentro de esta caja desde una aplicación externa como el explorador de archivos, y los datos se procesarán mediante el siguiente código.

```
var deposito;
function iniciar() {
  deposito = document.getElementById("deposito");
```

```
  deposito.addEventListener("dragenter", function(evento) {
    evento.preventDefault();
  });
  deposito.addEventListener("dragover", function(evento) {
    evento.preventDefault();
  });
  deposito.addEventListener("drop", soltar);
}
function soltar(evento) {
  evento.preventDefault();
  var archivos = evento.dataTransfer.files;
  var lista = "";
  for (var f = 0; f < archivos.length; f++) {
    lista += "Archivo: " + archivos[f].name + " " + archivos[f].size +
"<br>";
  }
  deposito.innerHTML = lista;
}
window.addEventListener("load", iniciar);
```

Listado 17-10: *Procesando los datos contenidos en la propiedad* `files`

El código del Listado 17-10 lee la propiedad **`files`** del objeto **`DataTransfer`** y crea un bucle para mostrar el nombre y el tamaño de cada archivo que se ha soltado en la caja. La propiedad **`files`** devuelve un array de objetos **`File`**, por lo que debemos leer las propiedades **`name`** y **`size`** de cada objeto para obtener esta información (ver objetos **`File`** del Capítulo 16).

Hágalo usted mismo: cree nuevos archivos con los códigos de los Listados 17-9 y 17-10, y abra el documento en su navegador. Arrastre archivos desde el explorador de archivos o cualquier otro programa similar dentro de la caja de la izquierda. Debería ver una lista con los nombres y tamaños de cada uno de los archivos que se han soltado dentro de la caja.

Capítulo 18
API Geolocation

18.1 Ubicación geográfica

La API Geolocation se diseñó para ofrecer un mecanismo de detección estándar a los navegadores con el que los desarrolladores pudieran determinar la ubicación geográfica del usuario. Anteriormente, solo teníamos la opción de construir una extensa base de datos con direcciones IP y programar aplicaciones de alto consumo en el servidor que solo nos daban una idea aproximada de la ubicación del usuario (a veces tan imprecisa como el país). Esta API aprovecha nuevos sistemas, como triangulación de redes y GPS, para devolver una ubicación precisa del dispositivo que está ejecutando la aplicación. La información obtenida se puede usar para crear aplicaciones que se adaptan a la ubicación del usuario o facilitar información localizada de forma automática. Se incluyen tres métodos en la API con este propósito.

getCurrentPosition(ubicación, error, configuración**)**—Este método se usa para solicitudes individuales. Acepta hasta tres atributos: una función para procesar la ubicación obtenida, una función para procesar errores y un objeto para configurar cómo se adquirirá la información (solo se requiere el primer atributo).

watchPosition(ubicación, error, configuración**)**—Este método es similar al método anterior excepto porque controla la ubicación constantemente para detectar e informar de cualquier cambio ocurrido. Funciona de forma similar al método **`setInterval()`**, repitiendo el proceso de forma automática en un periodo de tiempo según los valores declarados por defecto o la configuración especificada por los atributos.

clearWatch(identificador**)**—Este método detiene el proceso comenzado por el método **`watchPosition()`**. Es similar al método **`clearInterval()`** que se usa para detener el proceso iniciado por **`setInterval()`**.

El siguiente va a ser nuestro documento para este capítulo. Es un documento sencillo con solo un botón dentro de un elemento **`<section>`** que vamos a usar para mostrar la información que obtiene el sistema de localización.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Geolocation</title>
  <script src="geolocation.js"></script>
</head>
<body>
  <section id="ubicacion">
    <button type="button" id="obtener">Obtener mi ubicacion</button>
  </section>
</body>
</html>
```

Listado 18-1: *Creando un documento para probar la API Geolocation*

Obteniendo la ubicación

Como hemos mencionado antes, solo necesitamos asignar un atributo al método **getCurrentPosition()** para obtener la ubicación. Este atributo es una función que recibe un objeto llamado **Position** con toda la información que obtiene el sistema de localización. El objeto **Position** tiene dos propiedades para almacenar los valores.

coords—Esta propiedad contiene otro objeto con un grupo de propiedades que devuelven la posición del dispositivo. Las propiedades disponibles son **latitude**, **longitude**, **altitude** (en metros), **accuracy** (en metros), **altitudeAccuracy** (en metros), **heading** (en grados) y **speed** (en metros por segundo).

timestamp—Esta propiedad indica la hora en la que se ha requerido la información.

La API se define en un objeto llamado **Geolocation**. El objeto **Navigator** incluye la propiedad **geolocation** para ofrecer acceso a este objeto. Para acceder a los métodos de esta API, tenemos que llamarlos desde la propiedad **geolocation** de la propiedad **navigator** del objeto **Window**, como en **navigator.geolocation.getCurrentPosition()**. El siguiente ejemplo implementa esta sintaxis para obtener la ubicación actual del usuario con el método **getCurrentPosition()**.

```
function iniciar() {
  var elemento = document.getElementById("obtener");
  elemento.addEventListener("click", obtenerubicacion);
}
function obtenerubicacion() {
  navigator.geolocation.getCurrentPosition(mostrar);
}
function mostrar(posicion) {
  var ubicacion = document.getElementById("ubicacion");
  var datos = "";
  datos += "Latitud: " + posicion.coords.latitude + "<br>";
  datos += "Longitud: " + posicion.coords.longitude + "<br>";
  datos += "Exactitud: " + posicion.coords.accuracy + "mts.<br>";
  ubicacion.innerHTML = datos;
}
window.addEventListener("load", iniciar);
```

Listado 18-2: *Obteniendo la información de la ubicación*

En el código del Listado 18-2 hemos definido una función llamada **mostrar()** para procesar la información generada por el método **getCurrentPosition()**. Cuando se llama a este método, se crea un nuevo objeto **Position** con la información actual y se envía a la función **mostrar()**. Dentro de la función, referenciamos ese objeto con la variable **posicion** y luego usamos esta variable para mostrar los datos al usuario.

El objeto **Position** contiene dos propiedades importantes: **coords** y **timestamp**. En nuestro ejemplo, usamos **coords** para acceder a la información que queremos (latitud, longitud y precisión). Estos valores se almacenan en la variable **datos** y luego se muestran en la pantalla como el nuevo contenido del elemento **ubicacion**.

Hágalo usted mismo: cree dos archivos con los códigos de los Listados 18-1 y 18-2, y abra el documento en su navegador. Cuando haga clic en el botón, el navegador le preguntará si desea activar el sistema de localización para esta aplicación. Si autoriza a la aplicación a acceder a esta información, los valores de la latitud, longitud y la precisión de estos datos se mostrarán en la pantalla (puede tardar unos segundos en estar disponible).

Agregando un segundo atributo (otra función) al método **`getCurrentPosition()`** podemos capturar los errores producidos en el proceso, como el error que ocurre cuando el usuario le niega a nuestra aplicación el acceso al sistema de localización.

Junto al objeto **`Position`**, el método **`getCurrentPosition()`** devuelve el objeto **`PositionError`** si se detecta un error. El objeto contiene dos propiedades, **`error`** y **`message`**, para facilitar el valor y una descripción del error. Los tres posibles errores se representan mediante constantes.

PERMISSION_DENIED—Valor 1. Este error ocurre cuando el usuario niega a la API Geolocation acceso a la información de su ubicación.

POSITION_UNAVAILABLE—Valor 2. Este error ocurre cuando no se puede determinar la posición del dispositivo.

TIMEOUT—Valor 3. Este error ocurre cuando la posición no se puede determinar en el periodo de tiempo declarado en la configuración.

Si queremos informar de errores, solo tenemos que crear una nueva función y asignarla como el segundo parámetro del método **`getCurrentPosition()`**.

```
function iniciar() {
  var elemento = document.getElementById("obtener");
  elemento.addEventListener("click", obtenerubicacion);
}
function obtenerubicacion() {
  navigator.geolocation.getCurrentPosition(mostrar, mostrarerror);
}
function mostrar(posicion) {
  var ubicacion = document.getElementById("ubicacion");
  var datos = "";
  datos += "Latitud: " + posicion.coords.latitude + "<br>";
  datos += "Longitud: " + posicion.coords.longitude + "<br>";
  datos += "Exactitud: " + posicion.coords.accuracy + "mts.<br>";
  ubicacion.innerHTML = datos;
}
function mostrarerror(error) {
  alert("Error: " + error.code + " " + error.message);
}
window.addEventListener("load", iniciar);
```

Listado 18-3: *Mostrando mensajes de error*

Los mensajes de error están destinados a uso interno. El propósito es el de ofrecer un mecanismo para que la aplicación pueda reconocer la situación y proceder como corresponde. En el código del Listado 18-3 agregamos el segundo parámetro al método

getCurrentPosition() (otra función) y creamos esta función, llamada **mostrarerror()**, para mostrar los valores de las propiedades **code** y **message**. El valor de **code** será un entero entre **0** y **3** de según el número del error (listado anteriormente).

El objeto **PositionError** se envía a la función **mostrarerror()** y se representa mediante la variable **error**. También podemos controlar los errores individuales (**error.PERMISSION_DENIED**, por ejemplo) y actuar solo si esa condición en particular es **true** (verdadera).

El tercer valor posible para el método **getCurrentPosition()** es un objeto que contiene hasta tres propiedades.

enableHighAccuracy—Esta es una propiedad booleana que informa al sistema de que se requiere la ubicación más precisa posible. Cuando este valor se declara como **true**, el navegador intenta obtener la información a través de sistemas como GPS, por ejemplo, para determinar la ubicación exacta del dispositivo. Estos son sistemas de alto consumo y su uso se debe limitar a circunstancias específicas. Por esta razón, el valor por defecto de esta propiedad es **false**.

timeout—Esta propiedad indica el tiempo máximo que la operación puede tardar en completarse. Si la información no se adquiere dentro de este tiempo, se devuelve el error **TIMEOUT**. Su valor se expresa en milisegundos.

maximumAge—Las ubicaciones previas se preservan en un caché en el sistema. Si consideramos apropiado obtener la última información almacenada en lugar de una nueva (para evitar consumir recursos u obtener una respuesta rápida), esta propiedad se puede declarar con un tiempo límite. Si la última ubicación en el caché es más antigua que el valor de esta propiedad, el sistema determina una nueva ubicación. Su valor se expresa en milisegundos.

El siguiente código intenta obtener la ubicación más precisa posible en no más de 10 segundos, pero solo si no existe una ubicación previa en el caché capturada 60 segundos antes (si existe, ese será el objeto **Position** que devuelve).

```
function iniciar() {
  var elemento = document.getElementById("obtener");
  elemento.addEventListener("click", obtenerubicacion);
}
function obtenerubicacion() {
  var geoconfig = {
    enableHighAccuracy: true,
    timeout: 10000,
    maximumAge: 60000
  };
  navigator.geolocation.getCurrentPosition(mostrar, mostrarerror,
geoconfig);
}
function mostrar(posicion) {
  var ubicacion = document.getElementById("ubicacion");
  var datos = "";
  datos += "Latitud: " + posicion.coords.latitude + "<br>";
  datos += "Longitud: " + posicion.coords.longitude + "<br>";
  datos += "Exactitud: " + posicion.coords.accuracy + "mts.<br>";
  ubicacion.innerHTML = datos;
}
```

```
function mostrarerror(error) {
  alert("Error: " + error.code + " " + error.message);
}
window.addEventListener("load", iniciar);
```

Listado 18-4: *Configurando el sistema de localización*

En nuestro ejemplo, el objeto con la configuración se almacena en la variable **geoconfig**, y esta variable se declara como el tercer atributo del método **getCurrentPosition()**. No se ha modificado ningún otro aspecto en el resto del código con respecto al ejemplo anterior. La función **mostrar()** mostrará la información en la pantalla, independientemente de su origen (si proviene del caché o es nueva).

Desde este código podemos ver el propósito real de la API Geolocation y la razón por la que se ha introducido la API. Las características más útiles están destinadas a dispositivos móviles. Por ejemplo, el valor **true** para el atributo **enableHighAccuracy** sugiere al navegador que use sistemas como GPS para obtener la ubicación más precisa posible, el cual solo está disponible en estos tipos de dispositivos. También los métodos **watchPosition()** y **clearWatch()**, que estudiaremos a continuación, actualizan la ubicación permanentemente, aunque esto es solo útil cuando el dispositivo es móvil y se está moviendo. Esto plantea dos cuestiones. Primero, la mayoría de nuestros códigos se deben probar en un dispositivo móvil para ver cómo se desempeñarán en situaciones de la vida real. Y segundo, tenemos que ser responsables a la hora de usar esta API. Sistemas como GPS consumen muchos recursos y en la mayoría de los casos, el dispositivo se quedará sin batería si no tenemos cuidado.

Supervisando la ubicación

Al igual que el método **getCurrentPosition()**, el método **watchPosition()** recibe tres atributos y realiza la misma tarea: obtener la ubicación del dispositivo que está ejecutando la aplicación. La única diferencia es que el método **getCurrentPosition()** realiza una única operación mientras que **watchPosition()** ofrece nueva información de forma automática cada vez que cambia la ubicación. El método continúa supervisando la ubicación y envía información a la función cuando se detecta una nueva ubicación hasta que cancelemos el proceso con el método **clearWatch()**.

El método **watchPosition()** se implementa de la misma manera que el método **getCurrentPosition()**, tal como ilustra el siguiente ejemplo.

```
function iniciar() {
  var elemento = document.getElementById("obtener");
  elemento.addEventListener("click", obtenerubicacion);
}
function obtenerubicacion() {
  var geoconfig = {
    enableHighAccuracy: true,
    maximumAge: 60000
  };
  control = navigator.geolocation.watchPosition(mostrar, mostrarerror,
geoconfig);
}
function mostrar(posicion) {
  var ubicacion = document.getElementById("ubicacion");
```

```
  var datos = "";
  datos += "Latitud: " + posicion.coords.latitude + "<br>";
  datos += "Longitud: " + posicion.coords.longitude + "<br>";
  datos += "Exactitud: " + posicion.coords.accuracy + "mts.<br>";
  ubicacion.innerHTML = datos;
}
function mostrarerror(error) {
  alert("Error: " + error.code + " " + error.message);
}
window.addEventListener("load", iniciar);
```

Listado 18-5: *Probando el método* `watchPosition()`

No anotaremos ningún cambio si ejecutamos este ejemplo en un ordenador de escritorio, pero en un dispositivo móvil se mostrarán datos nuevos cada vez que la ubicación del dispositivo cambie. La frecuencia con la que se envía esta información a la función **`mostrar()`** se determina mediante el atributo **`maximumAge`**. Si la nueva ubicación se obtiene 60 segundos (60000 milisegundos) después de la anterior, se muestra; en caso contrario, no se llamará a la función **`mostrar()`**.

El valor que devuelve el método **`watchPosition()`** se almacena en la variable **`control`**. Esta variable actúa como un identificador de la operación. Si más adelante queremos cancelar el proceso, solo tenemos que ejecutar la instrucción **`clearWatch(control)`** y **`watchPosition()`** dejará de actualizar la información.

Google Maps

Hasta el momento hemos mostrado los datos de la ubicación exactamente como los recibimos. Sin embargo, estos valores no significan nada para la mayoría de las personas. No podemos establecer nuestra ubicación a partir de los valores de nuestra latitud y longitud, y mucho menos identificar una ubicación en el mundo desde estos valores. Contamos con dos alternativas: podemos usar la información internamente para calcular una posición, distancia y otras variables que nos permitan ofrecer información concreta a los usuarios (como productos o restaurantes en el área) o mostrar la información obtenida por la API Geolocation directamente al usuario en un formato más comprensible, como es un mapa.

Ya hemos mencionado antes en este libro la existencia de la API Google Maps. Esta es una API JavaScript externa de Google que no tiene nada que ver con HTML5, pero que se usa ampliamente en sitios web y aplicaciones modernas. La API ofrece una variedad de alternativas para trabajar con mapas interactivos e incluso vistas reales de ubicaciones específicas a través de la tecnología StreetView.

El siguiente es un ejemplo simple que implementa una parte de esta API llamada *Static Maps API*. Con esta API podemos construir una URL con la información de la ubicación, y nos devuelve un mapa del área seleccionada.

```
function iniciar() {
  var elemento = document.getElementById("obtener");
  elemento.addEventListener("click", obtenerubicacion);
}
function obtenerubicacion() {
  navigator.geolocation.getCurrentPosition(mostrar, mostrarerror);
}
```

```
function mostrar(posicion) {
  var ubicacion = document.getElementById("ubicacion");
  var mapurl = "http://maps.google.com/maps/api/staticmap?center=" +
posicion.coords.latitude + "," + posicion.coords.longitude +
"&zoom=12&size=400x400&sensor=false&markers=" +
posicion.coords.latitude + "," + posicion.coords.longitude;
  ubicacion.innerHTML = '<img src="' + mapurl + '">';
}
function mostrarerror(error) {
  alert("Error: " + error.code + " " + error.message);
}
window.addEventListener("load", iniciar);
```

Listado 18-6: *Representando la ubicación en un mapa con la API Google Maps*

La aplicación es sencilla. Usamos el método **getCurrentPosition()** y enviamos la información a la función **mostrar()** como siempre, pero ahora en esta función los valores del objeto **Position** se agregan a una URL de Google y luego la dirección se usa como fuente de un elemento **<img>** para mostrar la imagen que devuelve Google en la pantalla.

Figura 18-1: *Imagen que devuelve la API Google Maps*

Hágalo usted mismo: pruebe el código del Listado 18-6 en su navegador usando el documento del Listado 18-1. Debería ver un mapa con su ubicación en la pantalla. Cambie los valores de los atributos **zoom** y **size** en la URL para modificar el mapa que devuelve la API. Visite la página web de la API Google Maps para encontrar otras alternativas: https://developers.google.com/maps/.

Capítulo 19

API History

19.1 Historial

El historial del navegador es una lista de todas las páginas web (URL) que visita el usuario en una sesión. Es lo que hace posible la navegación. Si usamos los botones de navegación a la izquierda o la derecha de la barra de navegación en cada navegador, podemos movernos a través de esta lista y cargar documentos que hemos visitado con anterioridad.

Navegación

Con las flechas del navegador podemos cargar una página web que hemos visitado anteriormente o volver a la última, pero a veces es útil poder navegar a través del historial del navegador desde dentro del documento. Para este propósito, los navegadores incluyen la API History. Esta API incluye propiedades y métodos para manipular el historial y gestionar la lista de URL que contiene. Las siguientes son las propiedades y métodos disponibles para simular los botones de navegación desde código JavaScript.

length—Esta propiedad devuelve el número de entradas en el historial (el total de URL en la lista).

back()—Este método lleva al navegador un lugar hacia atrás en el historial (emulando la flecha izquierda).

forward()—Este método lleva al navegador un lugar hacia adelante en el historial (emulando la flecha derecha).

go(pasos**)**—Este método lleva al navegador hacia adelante o hacia atrás en el historial los pasos que especifica el atributo. El valor del atributo puede ser positivo o negativo según la dirección que elegimos.

La API la define el objeto `History`, que es accesible desde una propiedad del objeto `Window` llamada `history`. Cada vez que queremos leer las propiedades de la API o llamar a sus métodos, tenemos que hacerlo desde esta propiedad, como en `window.history.back()` o `history.back()` (la propiedad `window` se puede ignorar, tal como hemos explicado en el Capítulo 6).

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API History</title>
  <script>
    function volver() {
      window.history.back();
    }
```

```
  </script>
</head>
<body>
  <section>
    <button type="button" onclick="volver()">Volver a la página
anterior</button>
  </section>
</body>
</html>
```

Listado 19-1: *Navegando hacia atrás en el historial del navegador*

El documento del Listado 19-1 ilustra lo sencillo que resulta implementar estos métodos. El documento define un botón que llama a la función **`volver()`** cuando el usuario hace clic en el mismo. En esta función llamamos al método **`back()`** de la API History para llevar al usuario a la página web anterior.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 19-1. Visite un sitio web que conozca y luego cargue el documento en su navegador. Haga clic en el botón. El navegador debería cargar el sitio web anterior y mostrarlo en la pantalla.

URL

Como veremos en el Capítulo 21, estos días es común programar pequeñas aplicaciones que obtienen información desde un servidor y la muestran en la pantalla dentro del documento actual sin actualizar la página o cargar una nueva. Los usuarios interactúan con los sitios web y las aplicaciones desde la misma URL, recibiendo información, introduciendo datos y obteniendo los resultados impresos en la misma página. Sin embargo, los navegadores siguen la actividad del usuario a través de las URL. Las URL son los datos dentro de la lista de navegación, las direcciones que indican dónde está ubicado el usuario. Debido a que las nuevas aplicaciones web evitan usar URL para cargar nueva información, en el proceso se pierden pasos importantes. Los usuarios pueden actualizar datos en una página web docenas de veces sin dejar ningún rastro en el historial del navegador que nos indique los pasos seguidos y nos ayude a volver hacia atrás. Para solucionar este problema, la API History incluye propiedades y métodos que modifican la URL en la barra de navegación, así como el historial del navegador. Las siguientes son las que más se usan.

state—Esta propiedad devuelve el valor del estado de la entrada actual.

pushState(estado, título, url**)**—Este método crea una nueva entrada en el historial del navegador. El atributo **estado** declara un valor para el estado de la entrada. Es útil para identificar la entrada más adelante y se puede especificar como una cadena de caracteres o un objeto JSON. El atributo **título** es el título de la entrada y el atributo **url** es la URL para la entrada que estamos generando (este valor reemplazará la URL actual en la barra de navegación).

replaceState(estado, título, url**)**—Este método trabaja igual que **`pushState()`**, pero no genera una nueva entrada. En su lugar, reemplaza la información de la entrada actual.

Si queremos crear una nueva entrada en el historial del navegador y cambiar la URL dentro de la barra de navegación, tenemos que usar el método **pushState()**. El siguiente ejemplo muestra cómo funciona.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API History</title>
  <link rel="stylesheet" href="history.css">
  <script src="history.js"></script>
</head>
<body>
  <section id="contenidoprincipal">
    <p>Este contenido nunca es actualizado</p>
    <p><span id="url">página 2</span></p>
  </section>
  <aside id="cajadatos"></aside>
</body>
</html>
```

Listado 19-2: *Creando un documento básico para experimentar con la API History*

En este documento hemos incluido contenido permanente dentro de un elemento **<section>** identificado con el nombre **contenidoprincipal**, un texto que convertiremos en un enlace para generar una página virtual del sitio web, y la tradicional **cajadatos** para el contenido alternativo. Los siguientes son los estilos para el documento.

```
#contenidoprincipal {
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos {
  float: left;
  width: 500px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
#contenidoprincipal span {
  color: #0000FF;
  cursor: pointer;
}
```

Listado 19-3: *Definiendo los estilos para las cajas y los elementos* <span>

Lo básico: la propiedad **cursor** es una propiedad CSS que se usa para especificar el gráfico que representará el puntero del ratón. El sistema cambia este cursor automáticamente de acuerdo con el contenido que se encuentra debajo del puntero. Para contenido general, como elementos estructurales o imágenes, el puntero se representa con una flecha, para

texto es una barra vertical y para enlaces se muestra como una pequeña mano. En el Listado 19-3 hemos convertido el puntero en una mano para indicar al usuario que puede hacer clic en el contenido de los elementos **<span>**. Hay varios valores disponibles para esta propiedad. Para obtener más información, visite nuestro sitio web y siga los enlaces de este capítulo.

Lo que vamos a hacer en este ejemplo es agregar una nueva entrada con el método **pushState()** y actualizar el contenido sin cargar nuevamente la página o descargar una nueva.

```
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  url = document.getElementById("url");
  url.addEventListener("click", cambiarpagina);
}
function cambiarpagina() {
  cajadatos.innerHTML = "La url es página2";
  history.pushState(null, null, "pagina2.html");
}
window.addEventListener("load", iniciar);
```

Listado 19-4: *Generando una nueva URL y contenido*

En la función **iniciar()** del Listado 19-4 hemos creado la referencia apropiada para la **cajadatos** y agregado un listener para el evento **click** al elemento **<span>**. Cada vez que el usuario hace clic en el texto dentro del elemento **<span>,** se llama a la función **cambiarpagina()**. Esta función realiza dos tareas: actualiza el contenido de la página con nueva información e inserta una nueva URL en el historial del navegador. Después de que se ejecuta la función, la **cajadatos** muestra el texto "La url es página2", y la URL del documento principal en la barra de navegación se reemplaza con la URL pagina2.html.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 19-2, un archivo CSS llamado history.css con los estilos del Listado 19-3 y un archivo JavaScript llamado history.js con el código del Listado 19-4. Suba todos los archivos a su servidor y abra el documento en su navegador. Haga clic en el texto "página 2" y vea cómo la URL en la barra de navegación cambia por la generada desde el código.

IMPORTANTE: las URL generadas desde estos métodos son URL falsas en el sentido de que los navegadores nunca comprueban la validez de estas direcciones y la existencia del documento al que apuntan. Es su responsabilidad asegurarse de que estas URL falsas son de hecho válidas y útiles.

La propiedad state

Lo que hemos hecho hasta el momento es manipular el historial del navegador. Le hemos hecho creer al navegador que el usuario ha visitado una URL que, en este momento, no existe. Después de que el usuario hace clic en el enlace "página 2", la URL falsa pagina2.html se ha mostrado en la barra de navegación y un nuevo contenido se ha insertado en la **cajadatos**, todo sin actualizar la página o cargar una nueva. Es un truco interesante, pero no completo. El

navegador aún no considera a la nueva URL como un documento real. Si vamos hacia atrás en el historial con los botones del navegador, la nueva URL reemplaza con la URL correspondiente al documento principal, pero el contenido del documento no se modifica. Necesitamos detectar cuándo se visitan se nuevo las URL falsas y realizar las modificaciones apropiadas en el documento para mostrar el contenido correspondiente.

Anteriormente hemos mencionado la existencia de la propiedad **`state`**. El valor de esta propiedad se puede declarar durante la generación de una nueva URL y esta es la manera en la que luego identificamos cuál es la URL actual. La API incluye el siguiente evento para trabajar con esta propiedad.

popstate—Este evento se desencadena cuando se visita de nuevo una URL o se carga el documento. Facilita la propiedad **`state`** con el valor del estado declarado cuando la URL se ha generado con los métodos **`pushState()`** o **`replaceState()`**. Este valor es **`null`** cuando la URL es real a menos que lo hayamos cambiado anteriormente por medio del método **`replaceState()`**, como veremos a continuación.

En el siguiente ejemplo, mejoramos el código anterior implementando el evento **`popstate`** y el método **`replaceState()`** para detectar cuál es la URL que el usuario está solicitando.

```
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  url = document.getElementById("url");
  url.addEventListener("click", cambiarpagina);
  window.addEventListener("popstate", nuevaurl);
  history.replaceState(1, null);
}
function cambiarpagina() {
  mostrarpagina(2);
  history.pushState(2, null, "pagina2.html");
}
function nuevaurl(evento) {
  mostrarpagina(evento.state);
}
function mostrarpagina(actual) {
  cajadatos.innerHTML = "La url es página " + actual;
}
window.addEventListener("load", iniciar);
```

***Listado 19-5:** Siguiendo la ubicación del usuario en el historial*

Tenemos que realizar dos tareas en nuestra aplicación para tener absoluto control de la situación. Primero, debemos declarar un valor de estado para cada URL que vamos a usar, las falsas y las reales. Y segundo, debemos actualizar el contenido del documento de acuerdo a la URL actual.

En la función **`iniciar()`** del Listado 19-5 se agrega un listener para el evento **`popstate`**. Cada vez que se visita de nuevo una URL, se ejecuta la función **`nuevaurl()`**. Esta función actualiza el contenido de **`cajadatos`** de acuerdo a la URL actual, toma el valor de la propiedad **`state`** y lo envía a la función **`mostrarpagina()`** para mostrarlo en pantalla.

Esto funciona por cada una de las URL falsas, pero como ya hemos explicado, las URL reales no tienen un valor de estado por defecto. Usando el método **`replaceState()`** al final de la

función **`iniciar()`** cambiamos la información de la entrada actual (la URL real del documento principal) y declaramos el valor 1 para su estado. Ahora cada vez que el usuario visita otra vez el documento principal, podemos detectarlo a partir de este valor.

La función **`cambiarpagina()`** es la misma excepto que esta vez usa la función **`mostrarpagina()`** para actualizar el contenido del documento y declarar el valor `2` para el estado de la URL falsa.

La aplicación funciona de la siguiente manera: cuando el usuario hace clic en el enlace "página 2", se muestra en pantalla el mensaje "La url es página 2" y la URL en la barra de navegación cambia a pagina2.html (incluida la ruta completa, por supuesto). Esto es lo que hemos hecho hasta el momento, pero aquí es donde las cosas se ponen interesante. Si el usuario hace clic en la flecha izquierda del navegador, la URL de la barra de navegación cambia a la anterior en el historial (esta es la URL real de nuestro documento) y se desencadena el evento **`popstate`**. Este evento llama a la función **`nuevaurl()`**, la cual lee el valor de la propiedad **`state`** y lo envía a la función **`mostrarpagina()`**. Ahora el valor del estado es 1 (el valor que declaramos para este URL usando el método **`replaceState()`**), y el mensaje que se muestra en pantalla es "La url es página 1". Si el usuario vuelve a cargar la URL falsa usando la flecha derecha del navegador, el valor del estado será `2` y el mensaje que se muestra en la pantalla será nuevamente "La url es página 2".

El valor de la propiedad **`state`** puede ser cualquier valor que necesitemos para identificar qué URL es la actual y adaptar el contenido del documento a la situación.

Hágalo usted mismo: utilice los archivos con los código de los Listados 19-2 y 19-3 para el documento HTML y estilos CSS. Copie el código del Listado 19-5 dentro del archivo history.js y suba los archivos a su servidor o servidor local. Abra el documento en su navegador y haga clic en el texto "página 2". La URL y el contenido de la **`cajadatos`** deberían cambiar de acuerdo a la URL correspondiente. Haga clic en los botones izquierdo y derecho del navegador varias veces para ver cómo cambia la URL en la barra de navegación y cómo se actualiza en pantalla el contenido asociado a esa URL.

Aplicación de la vida real

La siguiente es una aplicación más práctica. Vamos a usar la API History para cargar un grupo compuesto por cuatro imágenes desde el mismo documento. Cada imagen se asocia con una URL falsa que se puede usar luego para descargarla desde el servidor.

El documento principal se carga con una imagen por defecto. Esta imagen estará asociada al primero de cuatro enlaces que son parte del contenido permanente. Todos estos enlaces apuntarán a URL falsas que referencian un estado, no un documento real, incluido el enlace del documento principal que se cambiará a pagina1.html para preservar coherencia.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API History</title>
  <link rel="stylesheet" href="history.css">
  <script src="history.js"></script>
</head>
```

```
<body>
  <section id="contenidoprincipal">
    <p>Este contenido nunca es actualizado</p>
    <p><span id="url1">imagen 1</span></p>
    <p><span id="url2">imagen 2</span></p>
    <p><span id="url3">imagen 3</span></p>
    <p><span id="url4">imagen 4</span></p>
  </section>
  <aside id="cajadatos">
    <img id="imagen" src="monstruo1.gif">
  </aside>
</body>
</html>
```

Listado 19-6: *Creando el documento principal de nuestra aplicación*

La única diferencia entre esta nueva aplicación y la anterior es la cantidad de enlaces y nuevas URL que estamos generando. En el código del Listado 19-5 había dos estados: el estado 1 correspondiente al documento principal y el estado 2 para la URL falsa (pagina2.html) generado por el método **`pushState()`**. En este caso tenemos que automatizar el proceso y generar un total de cuatro URL falsas correspondientes a cada imagen disponible.

```
function iniciar() {
  for (var f = 1; f < 5; f++) {
    url = document.getElementById("url" + f);
    url.addEventListener("click", function(x){
      return function() {
        cambiarpagina(x);
      };
    }(f));
  }
  window.addEventListener("popstate", nuevaurl);
  history.replaceState(1, null, "pagina1.html");
}
function cambiarpagina(pagina) {
  mostrarpagina(pagina);
  history.pushState(pagina, null, "pagina" + pagina + ".html");
}
function nuevaurl(evento) {
  mostrarpagina(evento.state);
}
function mostrarpagina(actual) {
  if (actual != null) {
    var imagen = document.getElementById("imagen");
    imagen.src = "monstruo" + actual + ".gif";
  }
}
window.addEventListener("load", iniciar);
```

Listado 19-7: *Manipulando el historial*

En este ejemplo, usamos las mismas funciones anteriores. Primero el método **`replaceState()`** en la función **`iniciar()`** recibe el valor pagina1.html para su atributo **`url`**. Decidimos programar nuestra aplicación de esta manera, declarando el estado del documento

principal con el valor **1** y la URL como pagina1.html (independientemente de la URL real del documento) para ser coherentes con las demás URL. Esto facilita el proceso de movernos de una URL a otra porque podemos usar el mismo nombre junto con los valores de la propiedad **state** para construir cada URL. Podemos ver esto en práctica en la función **cambiarpagina()**. Cada vez que el usuario hace clic en uno de los enlaces del documento, esta función se ejecuta y la URL falsa se construye con el valor de la variable **pagina** y se agrega al historial. El valor que recibe esta función se ha declarado previamente en el bucle **for** al comienzo de la función **iniciar()**. Este valor se declara como **1** para el enlace "imagen 1", **2** para el enlace "imagen 2" y así sucesivamente.

Cada vez que se visita una URL, se ejecuta la función **mostrarpagina()** para actualizar la imagen de acuerdo a esa URL. Debido a que el evento **popstate** a veces se desencadena cuando la propiedad **state** es **null** (como cuando el documento principal se carga por primera vez), controlamos el valor recibido por la función **mostrarpagina()** antes de realizar otras tareas. Si este valor es diferente del valor **null**, significa que la propiedad **state** se ha definido para esa URL y la imagen que corresponde a ese estado se muestra en pantalla.

Las imágenes usadas para este ejemplo se han llamado monstruo1.gif, monstruo2.gif, monstruo3.gif y monstruo4.gif, siguiendo el mismo orden que los valores de la propiedad **state**. De este modo, usando este valor podemos seleccionar la imagen a mostrar.

Hágalo usted mismo: para probar el último ejemplo, utilice el documento HTML del Listado 19-6 con el código CSS del Listado 19-3. Copie el código del Listado 19-7 dentro del archivo history.js. Descargue los archivos monstruo1.gif, monstruo2.gif, monstruo3.gif y monstruo4.gif desde nuestro sitio web y suba todos los archivos a su servidor o servidor local. Abra el documento en su navegador y haga clic en los enlaces. Navegue a través de las URL que ha seleccionado usando los botones de navegación. Las imágenes en la pantalla deberían cambiar de acuerdo a la URL en la barra de navegación.

Lo básico: el bucle **for** usado en el código del Listado 19-7 para agregar un listener para el evento **click** a cada elemento **<span>** en el documento aprovecha la técnica descrita en el ejemplo del Listado 6-161, Capítulo 6. Para enviar el valor actual de la variable **f** a la función, tenemos que usar dos funciones anónimas. La primera función se ejecuta cuando se procesa el método **addEventListener()**. Esta función recibe el valor actual de la variable **f** (vea los paréntesis al final) y lo almacena en la variable **x**. Luego, la función devuelve una segunda función anónima con el valor de la variable **x**. Esta segunda función es la que se ejecutará cuando se desencadena el evento.

Capítulo 20
API Page Visibility

20.1 Visibilidad

Las aplicaciones web se están volviendo más sofisticadas y demandan recursos como nunca antes. Las páginas web ya no son documentos estáticos; JavaScript las ha convertido en aplicaciones completas, capaces de ejecutar procesos complejos sin interrupción, e incluso sin la intervención del usuario. Debido a esto, en algunos momentos estos procesos podrían requerir su cancelación o pausa para distribuir recursos y ofrecer una mejor experiencia al usuario. Con la intención de producir aplicaciones conscientes de su estado, los navegadores incluyen la API Page Visibility. Esta API informa a la aplicación acerca del estado actual de visibilidad del documento, por ejemplo, cuándo se oculta la pestaña o se minimiza la ventana, de modo que nuestro código pueda decidir qué hacer mientras nadie mira.

Estado

La API incluye una propiedad para informar del estado actual y un evento para permitir a la aplicación saber cuándo ha cambiado algo.

visibilityState—Esta propiedad devuelve el estado de visibilidad actual del documento. Los valores disponibles son **`hidden`** y **`visible`** (algunos navegadores también pueden haber implementado valores opcionales como **`prerender`** y **`unloaded`**).

visibilitychange—Este evento se desencadena cuando cambia el valor de la propiedad **`visibilityState`**.

La API es parte del objeto **`Document`** y, por lo tanto, es accesible desde la propiedad **`document`** del objeto **`Window`**. El siguiente documento implementa la propiedad y el evento provistos por la API para detectar cuándo abre el usuario otra pestaña y modifica el contenido de la página para reflejar el nuevo estado.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Page Visibility</title>
  <script>
    function iniciar() {
      document.addEventListener("visibilitychange", mostrarestado);
    }
    function mostrarestado() {
      var elemento = document.getElementById("aplicacion");
      elemento.innerHTML += "<br>" + document.visibilityState;
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
```

```
<body>
  <section id="aplicacion">
    <p>Abra otra pestaña o minimice esta ventana para cambiar el estado de visibilidad</p>
  </section>
</body>
</html>
```

***Listado 20-1:** Informando del estado de visibilidad*

En el código del Listado 20-1, se declara la función **`showstate()`** para responder al evento **`visibilitychange`**. Cuando el evento se desencadena, la función muestra el valor de la propiedad **`visibilityState`** en la pantalla para informar del estado actual del documento. Cuando la pestaña se reemplaza por otra pestaña o la ventana se minimiza, el valor de la propiedad **`visibilityState`** cambia a **`hidden`**, y cuando la pestaña o la ventana vuelven a ser visibles, el valor se declara nuevamente como **`visible`**.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 20-1 y abra el documento en su navegador. Abra una nueva pestaña o una pestaña abierta anteriormente. Regrese a la pestaña del documento. Debería ver las palabras hidden y visible impresas en la pantalla (el documento se ha ocultado cuando se ha abierto la otra pestaña y se ha vuelto visible cuando su pestaña se ha vuelto a abrir).

Una de las razones de la implementación de esta API es el alto consumo de recursos de aplicaciones modernas, pero la API también se diseñó como una forma de mejorar la experiencia del usuario. El siguiente ejemplo muestra cómo lograrlo con unas pocas líneas de código.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Page Visibility</title>
  <script>
    var video;
    function iniciar() {
      video = document.getElementById("medio");
      document.addEventListener("visibilitychange", mostrarestado);
      video.play();
    }
    function mostrarestado() {
      var estado = document.visibilityState;
      switch(estado) {
        case "visible":
          video.play();
          break;
        case "hidden":
          video.pause();
          break;
      }
    }
    window.addEventListener("load", iniciar);
```

```
  </script>
</head>
<body>
  <video id="medio" width="720" height="400">
    <source src="trailer.mp4">
    <source src="trailer.ogg">
  </video>
</body>
</html>
```

Listado 20-2: *Respondiendo al estado de visibilidad*

El código del Listado 20-2 reproduce un vídeo mientras el documento es visible y lo pausa cuando no lo es. Usamos las mismas funciones del ejemplo anterior, excepto que esta vez controlamos el valor de la propiedad **visibilityState** con una instrucción **switch** y se ejecutan los métodos **play()** o **pause()** para reproducir o pausar el vídeo según el estado actual.

Hágalo usted mismo: actualice el código de su archivo HTML con el documento del Listado 20-2. Descargue los archivos trailer.mp4 y trailer.ogg desde nuestro sitio web. Abra el nuevo documento en su navegador y alterne entre varias pestañas para ver cómo trabaja la aplicación. El vídeo se debería pausar cuando el documento no es visible y reproducir cuando es de nuevo visible.

Sistema de detección completo

Los navegadores no cambian el valor de la propiedad **visibilityState** cuando el usuario abre una nueva ventana. La API solo es capaz de detectar el cambio de visibilidad cuando la pestaña se oculta por otra o cuando la ventana se minimiza. Para determinar el estado de visibilidad en cualquier circunstancia, podemos complementar la API con los siguientes eventos que facilita el objeto **Window**.

blur—Este evento se desencadena cuando la ventana pierde foco (también se desencadena por los elementos).

focus—Este evento se desencadena cuando la ventana se enfoca de nuevo (también se desencadena por los elementos).

Si agregamos una pequeña función, podemos combinar todas las herramientas disponibles para construir un mejor sistema de detección.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Page Visibility</title>
  <script>
    var estado;
    function iniciar() {
      window.addEventListener("blur", function() {
        cambiarestado("hidden");
      });
```

```
      window.addEventListener("focus", function() {
        cambiarestado("visible");
      });
      document.addEventListener("visibilitychange", function() {
        cambiarestado(document.visibilityState);
      });
    }
    function cambiarestado(nuevoestado) {
      if (estado != nuevoestado) {
        estado = nuevoestado;
        mostrarestado();
      }
    }
    function mostrarestado() {
      var elemento = document.getElementById("aplicacion");
      elemento.innerHTML += "<br>" + estado;
    }
    window.addEventListener("load", iniciar);
  </script>
</head>
<body>
  <section id="aplicacion">
    <p>Abra otra ventana para cambiar el estado de visibilidad</p>
  </section>
</body>
</html>
```

Listado 20-3: *Combinando los eventos* `blur`*,* `focus` *y* `visibilitychange`

En la función **`iniciar()`** del Listado 20-3 agregamos listeners para los tres eventos. Se declara la función **`cambiarestado()`** para responder a los eventos y procesar el valor correspondiente al estado actual. Este valor queda determinado por cada uno de los eventos. El evento **`blur`** envía el valor **`hidden`**; el evento **`focus`** envía el valor **`visible`** y el evento **`visibilitychange`** envía el valor actual de la propiedad **`visibilityState`**. En consecuencia, la función **`cambiarestado()`** recibe el valor correcto, independientemente de cómo se ha ocultado el documento (por otra pestaña, ventana o programa). Usando la variable **`estado`**, la función compara el valor anterior con el nuevo, almacena el nuevo valor en la variable y llama a la función **`mostrarestado()`** para mostrar el estado en la pantalla.

Este proceso es algo más complicado que el anterior, pero considera todas las situaciones posibles en las que se puede ocultar el documento y el estado se modifica en toda ocasión.

Hágalo usted mismo: actualice el código en su archivo HTML con el documento del Listado 20-3. Abra el nuevo documento en su navegador, y alterne entre la ventana del navegador y otra ventana o programa. Debería ver las palabras hidden y visible impresas en la pantalla que reflejan los cambios de estado.

Capítulo 21
Ajax Level 2

21.1 El objeto XMLHttpRequest

En el antiguo paradigma los sitios web y aplicaciones accedían al servidor y ofrecían toda la información al mismo tiempo. Si se requería nueva información, el navegador tenía que acceder nuevamente al servidor y reemplazar la información actual con la nueva. Esto motivó el uso de la palabra *Páginas* para describir documentos HTML. Los documentos se reemplazaban por otros documentos como las páginas de un libro.

Este fue el mecanismo estándar para la Web hasta que alguien encontró un mejor uso para un antiguo objeto, introducido primero por Microsoft y mejorado luego por Mozilla, llamado **`XMLHttpRequest`**. Este objeto puede acceder al servidor y obtener información desde JavaScript sin actualizar o cargar un nuevo documento. En un artículo publicado en el año 2005, se asignó el nombre *Ajax* a este procedimiento.

Debido a la importancia de este objeto en aplicaciones modernas, los navegadores incluyen la API XMLHttpRequest Level 2 para el desarrollo de aplicaciones Ajax. Esta API incorpora características como la comunicación de origen cruzado y los eventos para controlar la evolución de la solicitud. Estas mejoras simplifican los códigos y ofrecen nuevas alternativas, como la posibilidad de interactuar con varios servidores desde la misma aplicación o trabajar con trozos pequeños de datos en lugar de archivos completos.

El elemento más importante de esta API es, por supuesto, el objeto **`XMLHttpRequest`**. Se incluye el siguiente constructor para crear este objeto.

XMLHttpRequest()—Este constructor devuelve un objeto **`XMLHttpRequest`** desde el cual podemos iniciar una solicitud y responder a eventos para controlar el proceso de comunicación.

El objeto creado por el constructor **`XMLHttpRequest()`** facilita algunos métodos para iniciar y controlar la solicitud.

open(método, url, asíncrona**)**—Este método configura una solicitud pendiente. El atributo **método** especifica el método HTTP que se usa para abrir la conexión (**`GET`** o **`POST`**), el atributo **url** declara la ubicación del código que va a procesar la solicitud y el atributo **asíncrona** es un valor booleano que declara el tipo de conexión, síncrona (**`false`**) o asíncrona (**`true`**). El método también puede incluir valores para definir el nombre de usuario y la clave cuando son necesarios.

send(datos**)**—Este método inicia la solicitud. El objeto **`XMLHttpRequest`** incluye varias versiones de este método para procesar diferentes tipos de datos. El atributo **datos** se puede omitir, declarar como un ArrayBuffer, un blob, un documento, una cadena de caracteres o un objeto **`FormData`**, como veremos más adelante.

abort()—Este método cancela la solicitud.

El siguiente ejemplo obtiene un archivo de texto desde el servidor usando el método **`GET`**.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Ajax Level 2</title>
  <link rel="stylesheet" href="ajax.css">
  <script src="ajax.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <button type="button" id="boton">Clic Aquí</button>
    </form>
  </section>
  <section id="cajadatos"></section>
</body>
</html>
```

Listado 21-1: *Creando un documento para procesar solicitudes Ajax*

Los siguientes son los estilos que necesitamos para diferenciar el formulario de la caja donde vamos a mostrar la información recibida desde el servidor.

```
#cajaformulario {
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos {
  float: left;
  width: 500px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
```

Listado 21-2: *Diseñando los estilos de las cajas en la pantalla*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 21-1, un archivo CSS llamado ajax.css con las reglas del Listado 21-2 y un archivo JavaScript llamado ajax.js para los códigos que se presentan a continuación. Para poder probar los ejemplos de este capítulo tiene que subir todos los archivos, incluidos el archivo JavaScript y el archivo al cual se envía la solicitud, a su servidor o su servidor local.

El código para nuestro primer ejemplo lee un archivo en el servidor y muestra su contenido en pantalla. Ningún dato se envía al servidor en esta oportunidad, por lo que solo necesitamos realizar una solicitud `GET` y mostrar la información que devuelve el servidor.

```
var cajadatos;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
```

```
  var boton = document.getElementById("boton");
  boton.addEventListener("click", leer);
}
function leer() {
  var url = "archivotexto.txt";
  var solicitud = new XMLHttpRequest();
  solicitud.addEventListener("load", mostrar);
  solicitud.open("GET", url, true);
  solicitud.send(null);
}
function mostrar(evento) {
  var datos = evento.target;
  if (datos.status == 200) {
    cajadatos.innerHTML = datos.responseText;
  }
}
window.addEventListener("load", iniciar);
```

Listado 21-3: *Leyendo un archivo*

En este ejemplo, la función **iniciar()** crea una referencia al elemento **cajadatos** y agrega un listener al botón para el evento **click**. Cuando se pulsa el botón, se ejecuta la función **leer()**. En esta función, se declara la URL del archivo a leer, y se crea un objeto con el constructor **XMLHttpRequest()** y asignado a la variable **solicitud**. Esta variable se usa para agregar un listener para el evento **load** a este objeto, y llamar a sus métodos **open()** y **send()** para configurar e iniciar la solicitud. Como no enviamos ningún dato en esta solicitud, el método **send()** se declara vacío (**null**), pero el método **open()** necesita sus atributos para configurar la solicitud; tenemos que declarar el tipo de solicitud como **GET**, especificar la URL del archivo que se va a leer y declarar el tipo de operación (**true** para asíncrona).

Una operación asíncrona significa que el navegador continuará procesando el resto del código mientras se está descargando el archivo. Se informa del final de la operación a través del evento **load**. Cuando se ha cargado el archivo, el evento se desencadena y se llama a la función **mostrar()**. Esta función controla el estado de la operación a través del valor de la propiedad **status** y luego inserta el valor de la propiedad **responseText** dentro de **cajadatos** para mostrar el contenido del archivo recibido en la pantalla.

Hágalo usted mismo: para probar este ejemplo, descargue el archivo archivotexto.txt desde nuestro sitio web. Suba este archivo y el resto de los archivos con los códigos de los Listados 21-1, 21-2 y 21-3 a su servidor o servidor local, y abra el documento en su navegador. Después de hacer clic en el botón, el contenido del archivo de texto se muestra en la pantalla.

Lo básico: los servidores devuelven información para informar del estado de la solicitud. Esta información se llama *HTTP status code* e incluye un número y un mensaje en el que se describe el estado. El evento **load** envía un objeto **ProgressEvent** a la función para informar del progreso y el estado de la solicitud. Este objeto incluye dos propiedades para obtener el estado de la solicitud: **status** y **statusText**. La propiedad **status** devuelve el número de estado y la propiedad **statusText** devuelve el mensaje. El valor 200, usado en el ejemplo del Listado 21-3, es uno de varios disponibles. Este valor indica que la solicitud se ha realizado correctamente (el mensaje de este

estado es "OK"). Si no se puede encontrar el recurso, el valor devuelto será 404. Para obtener una lista completa de códigos de estado HTTP, visite nuestro sitio web y siga los enlaces de este capítulo.

Propiedades

El objeto **XMLHttpRequest** incluye algunas propiedades para configurar la solicitud. Las siguientes son las que más se usan.

responseType—Esta propiedad declara el formato de los datos recibidos. Acepta cinco valores diferentes: **text**, **arraybuffer**, **blob**, **document**, o **json**.

timeout—Esta propiedad determina el período de tiempo máximo permitido para que se procese la solicitud. Acepta un valor en milisegundos.

También contamos con tres tipos diferentes de propiedades que podemos usar para obtener la información que devuelve la solicitud.

response—Esta es una propiedad de propósito general. Devuelve la respuesta a la solicitud en el formato especificado por el valor de la propiedad **responseType**.

responseText—Esta propiedad devuelve la respuesta a la solicitud como texto.

responseXML—Esta propiedad devuelve la respuesta a la solicitud como datos XML.

La propiedad más útil es **response**. Esta propiedad devuelve los datos en el formato declarado previamente por la propiedad **responseType**. El siguiente ejemplo obtiene una imagen desde el servidor con estas propiedades.

```
var cajadatos;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  var boton = document.getElementById("boton");
  boton.addEventListener("click", leer);
}
function leer() {
  var url = "miimagen.jpg";
  var solicitud = new XMLHttpRequest();
  solicitud.responseType = "blob";
  solicitud.addEventListener("load", mostrar);
  solicitud.open("GET", url, true);
  solicitud.send(null);
}
function mostrar(evento) {
  var datos = evento.target;
  if (datos.status == 200) {
    var imagen = URL.createObjectURL(datos.response);
    cajadatos.innerHTML = '<img src="' + imagen + '">';
  }
}
window.addEventListener("load", iniciar);
```

Listado 21-4: *Leyendo una imagen en el servidor*

En el Listado 21-4 la propiedad **responseType** de la solicitud se declara como **blob**. Ahora los datos recibidos se procesarán como un blob que podemos almacenar en un archivo, cortar, subir nuevamente al servidor, etc. En este ejemplo, lo usamos como fuente de un elemento **<img>** para mostrar la imagen en pantalla. Para convertir el blob en una URL para la fuente de la imagen, aplicamos el método **createObjectURL()** introducido en el Capítulo 16.

Eventos

Además de **load**, la API incluye los siguientes eventos para el objeto **XMLHttpRequest**.

loadstart—Este evento se desencadena cuando se inicia la solicitud.

progress—Este evento se desencadena periódicamente mientras se reciben o se suben los datos.

abort—Este evento se desencadena cuando se anula la solicitud.

error—Este evento se desencadena cuando ocurre un error durante la solicitud.

load—Este evento se desencadena cuando la solicitud se ha completado.

timeout—Si se ha especificado un valor **timeout**, este evento se disparará cuando la solicitud no se pueda completar en el período de tiempo especificado.

loadend—Este evento se desencadena cuando la solicitud se ha completado (sin importar si se ha realizado correctamente o no).

El evento más útil de todos es **progress**. Este evento se desencadena aproximadamente cada 50 milisegundos para informar al código acerca del estado de la solicitud. Aprovechando el evento **progress,** podemos notificar al usuario de cada paso del proceso y crear una aplicación de comunicación profesional.

```
var cajadatos;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  var boton = document.getElementById("boton");
  boton.addEventListener("click", leer);
}
function leer() {
  var url = "trailer.ogg";
  var solicitud = new XMLHttpRequest();
  solicitud.addEventListener("loadstart", comenzar);
  solicitud.addEventListener("progress", estado);
  solicitud.addEventListener("load", mostrar);
  solicitud.open("GET", url, true);
  solicitud.send(null);
}
function comenzar() {
  var progreso = document.createElement("progreso");
  progreso.value = 0;
  progreso.max = 100;
  progreso.innerHTML = "0%";
  cajadatos.appendChild(progreso);
}
```

```
function estado(evento) {
  if (evento.lengthComputable) {
    var porcentaje = Math.ceil(evento.loaded / evento.total * 100);
    var progreso = cajadatos.querySelector("progreso");
    progreso.value = porcentaje;
    progreso.innerHTML = porcentaje + '%';
  } else {
    console.log("El tamaño no puede ser calculado");
  }
}
function mostrar(evento) {
  var datos = evento.target;
  if (datos.status == 200) {
    cajadatos.innerHTML = "Terminado";
  }
}
window.addEventListener("load", iniciar);
```

Listado 21-5: *Mostrando el progreso de la solicitud*

En el Listado 21-5 el código responde a tres eventos, **loadstart**, **progress** y **load**, para controlar la solicitud. El evento **loadstart** llama a la función **start()** para mostrar el progreso en la pantalla por primera vez. Mientras se carga el archivo, el evento **progress** ejecuta la función **estado()**. Esta función calcula el progreso a partir de los valores que devuelven las propiedades del objeto **ProgressEvent** (comentado en el Capítulo 16). Si la propiedad **lengthComputable** devuelve **true**, y por tanto el sistema ha podido determinar los valores, calculamos el porcentaje de progreso con la formula **evento.loaded / evento.total * 100**.

Finalmente, cuando el archivo se ha descargado por completo, se desencadena el evento **load** y la función **mostrar()** muestra el texto "Terminado" en la pantalla.

Hágalo usted mismo: para poder ver cómo trabaja la barra de progreso, deberá descargar archivos extensos. En el código del Listado 21-5, hemos cargado el vídeo trailer.ogg, utilizado en capítulos anteriores, pero este archivo se podría cargar demasiado rápido y no permitir a la barra de progreso informar del estado del proceso. Además, algunos servidores no informan al navegador sobre el tamaño del archivo. En casos como estos, la propiedad **lengthComputable** devuelve el valor **false**. En nuestro ejemplo, solo imprimimos un mensaje en la consola cuando esto sucede, pero debería ofrecer una mejor respuesta que permita al usuario saber qué está ocurriendo.

Enviando datos

Del mismo modo que podemos recibir datos desde el servidor con Ajax, también podemos enviarlos. Enviar datos con el método **GET** es tan sencillo como incluir los valores en la URL. Todo lo que tenemos que hacer es insertar los valores en la URL, como explicamos en el Capítulo 2, y estos se envían junto con la solicitud (por ejemplo, miarchivo.php?var1=25&var2=46). Pero el método **GET** presenta limitaciones, especialmente en la cantidad de datos permitidos. Una alternativa es usar el método **POST**. A diferencia de la solicitud realizada con el método **GET**, una solicitud **POST** incluye el cuerpo del mensaje que nos permite enviar cualquier tipo de información al servidor y del tamaño que necesitemos.

Un formulario HTML es normalmente la mejor manera de facilitar esta información, pero en aplicaciones dinámicas, esta no es la mejor opción o la más apropiada. La API incluye el objeto `FormData` para resolver este problema. Este objeto crea un formulario virtual que podemos completar con los valores que queremos enviar al servidor. La API incluye un constructor para obtener este objeto y un método para agregarle datos.

FormData(formulario**)**—Este constructor devuelve un objeto `FormData`. El atributo **formulario** es una referencia a un formulario, lo que ofrece una forma simple de incluir un formulario HTML completo dentro del objeto (opcional).

append(nombre, valor**)**—Este método agrega datos a un objeto `FormData`. Acepta pares nombre/valor como atributos. El atributo **nombre** es el nombre que queremos usar para identificar el valor y el atributo **valor** es el valor mismo (puede ser una cadena de caracteres o un blob). Los datos que devuelve este método representan un campo de formulario.

El siguiente ejemplo crea un formulario pequeño con dos campos de texto llamados `nombre` y `apellido`. El formulario se envía al servidor, procesado, y la respuesta se muestra en pantalla.

```
var cajadatos;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  var boton = document.getElementById("boton");
  boton.addEventListener("click", enviar);
}
function enviar() {
  var datos = new FormData();
  datos.append("nombre", "Juan");
  datos.append("apellido", "Perez");
  var url = "procesar.php";
  var solicitud = new XMLHttpRequest();
  solicitud.addEventListener("load", mostrar);
  solicitud.open("POST", url, true);
  solicitud.send(datos);
}
function mostrar(evento) {
  var datos = evento.target;
  if (datos.status == 200) {
    cajadatos.innerHTML = datos.responseText;
  }
}
window.addEventListener("load", iniciar);
```

Listado 21-6: *Enviando un formulario virtual al servidor*

Cuando la información se envía al servidor, es con el propósito de procesarla y producir un resultado. Normalmente, este resultado se almacena en el servidor y se devuelven algunos datos para ofrecer una respuesta. En el ejemplo del Listado 21-6 enviamos los datos al archivo procesar.php y mostramos la información que devuelve el código de este archivo en la pantalla. Los archivos con la extensión .php contienen código PHP, que es código que se ejecuta en el servidor. En este libro no explicamos cómo programar una aplicación que funciona en el servidor, pero para completar este ejemplo vamos a crear un código sencillo que devuelve al navegador un documento con los valores enviados por la aplicación.

```
<?php
  print('Su nombre es: '.$_POST['nombre'].'<br>');
  print('Su apellido es: '.$_POST['apellido']);
?>
```

Listado 21-7: *Respondiendo a una solicitud* `POST` *(procesar.php)*

Veamos primero cómo se va a enviar y preparar la información. En la función **`enviar()`** del Listado 21-6 se llama al constructor **`FormData()`** y el objeto **`FormData`** que se devuelve se almacena en la variable **`datos`**. Se agregan dos pares nombre/valor a este objeto con los nombres **`nombre`** y **`apellido`** mediante el método **`append()`**.

La inicialización de la solicitud es igual que en ejemplos anteriores, excepto que esta vez el primer atributo del método **`open()`** es **`POST`** en lugar de **`GET`**, y el atributo del método **`send()`** es el objeto **`FormData`** que acabamos de crear.

Cuando se pulsa el botón del documento, se llama a la función **`enviar()`** y el formulario creado por el objeto **`FormData`** se envía al servidor. El archivo procesar.php del servidor lee los campos del formulario (**`nombre`** y **`apellido`**) y devuelve un documento al navegador que contiene estos valores. Cuando nuestro código JavaScript recibe esta información, se ejecuta la función **`mostrar()`** y el contenido de ese documento se muestra en pantalla.

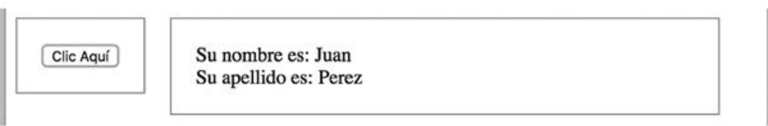

Figura 21-1: *Respuesta desde el servidor recibida por la aplicación*

Hágalo usted mismo: este ejemplo requiere que se suban varios archivos al servidor. Entre ellos se deben incluir el documento HTML y estilos CSS de los Listados 21-1 y 21-2. El código JavaScript del Listado 21-6 reemplaza al anterior. También tiene que crear un nuevo archivo llamado procesar.php con el código del Listado 21-7 para responder a la solicitud. Suba todos los archivos a su servidor y abra el documento HTML en su navegador. Después de hacer clic en el botón Clic Aquí, debería ver el texto que devuelve el archivo procesar.php en la pantalla, tal como ilustra la Figura 21-1.

Subiendo archivos

Subir archivos a un servidor es actualmente una de las mayores preocupaciones de los desarrolladores web debido a que es una función que requieren la mayoría de las aplicaciones en Internet, pero no había sido considerada por los navegadores hasta la introducción de HTML5. Esta API se encarga de esta situación e incorpora una nueva propiedad que devuelve un objeto **`XMLHttpRequestUpload`**. Este objeto facilita todas las propiedades, métodos y eventos disponibles en el objeto **`XMLHttpRequest`**, pero se ha diseñado para controlar el proceso de subir archivos al servidor.

upload—Esta propiedad devuelve un objeto **`XMLHttpRequestUpload`**. La propiedad se debe llamar desde un objeto **`XMLHttpRequest`**.

Para demostrar cómo trabaja este objeto, vamos a crear un nuevo documento HTML con un campo **<input>** con el que seleccionaremos el archivo que queremos subir al servidor.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Ajax Level 2</title>
  <link rel="stylesheet" href="ajax.css">
  <script src="ajax.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <label for="archivos">Archivo a subir: </label>
      <input type="file" name="archivos" id="archivos">
    </form>
  </section>
  <section id="cajadatos"></section>
</body>
</html>
```

Listado 21-8: *Creando un documento para subir archivos*

Cuando queremos enviar un archivo a un servidor, tenemos que enviar el objeto **File** que representa el archivo, por lo que podemos usar un objeto **FormData** para este propósito. El sistema detecta automáticamente el tipo de información agregada a un objeto **FormData** y crea las cabeceras apropiadas para la solicitud. El resto del proceso es el mismo que hemos estudiado antes en este capítulo.

```
var cajadatos;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  var archivos = document.getElementById("archivos");
  archivos.addEventListener("change", subir);
}
function subir(evento) {
  var archivos = evento.target.files;
  var archivo = archivos[0];

  var datos = new FormData();
  datos.append("archivo", archivo);

  var url = "procesar.php";
  var solicitud = new XMLHttpRequest();
  solicitud.addEventListener("loadstart", comenzar);
  solicitud.addEventListener("load", mostrar);
  var xmlupload = solicitud.upload;
  xmlupload.addEventListener("progress", estado);
  solicitud.open("POST", url, true);
  solicitud.send(datos);
}
```

```
function comenzar() {
  var progreso = document.createElement("progress");
  progreso.value = 0;
  progreso.max = 100;
  progreso.innerHTML = "0%";
  cajadatos.appendChild(progreso);
}
function estado(evento) {
  if (evento.lengthComputable) {
    var porcentaje = parseInt(evento.loaded / evento.total * 100);
    var progreso = cajadatos.querySelector("progress");
    progreso.value = porcentaje;
    progreso.innerHTML = porcentaje + '%';
  } else {
    console.log("El tamaño no puede ser calculado");
  }
}
function mostrar(evento) {
  var datos = evento.target;
  if (datos.status == 200) {
    cajadatos.innerHTML = "Terminado";
  }
}
window.addEventListener("load", iniciar);
```

Listado 21-9: *Subiendo un archivo con* `FormData()`

La función principal del código del Listado 21-9 es **`subir()`**. Se llama a la función cuando el usuario selecciona un nuevo archivo desde el elemento **`<input>`** en el documento (cuando se desencadena el evento **`change`**). El archivo seleccionado se recibe y almacena en la variable **`archivo`**, exactamente igual a lo que hemos hecho con la API File en el Capítulo 16 y también con la API Drag and Drop en el Capítulo 17. Una vez que tenemos la referencia al archivo, se crea el objeto **`FormData`** y el archivo se agrega al objeto por medio del método **`append()`**. Para enviar este formulario, iniciamos una solicitud **`POST`** y declaramos todos los listeners para la solicitud, excepto **`progress`**. El evento **`progress`** se usa para controlar el proceso mientras se sube el archivo al servidor, por lo que primero tenemos que leer la propiedad **`upload`** para obtener el objeto **`XMLHttpRequestUpload`**. Una vez obtenido este objeto, finalmente podemos agregar el listener para el evento **`progress`** y enviar la solicitud.

El resto del código muestra una barra de progreso en la pantalla cuando se inicia el proceso y actualiza esta barra de acuerdo a su progreso.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 21-8 y un archivo JavaScript llamado ajax.js con el código del Listado 21-9. También necesitará un archivo CSS llamado ajax.css con los estilos del Listado 21-2. Suba los archivos a su servidor o a un servidor local. Abra el documento en su navegador y presione el botón para seleccionar un archivo. Cargue un archivo extenso para poder ver cómo evoluciona la barra de progreso.

IMPORTANTE: algunos servidores establecen un límite en el tamaño de los archivos que se pueden subir. Por defecto, en la mayoría de los casos, este tamaño es de solo unos 2 megabytes. Los archivos más grandes se rechazarán. Puede modificar este comportamiento desde los archivos de configuración del servidor (por ejemplo, **`php.ini`**).

Aplicación de la vida real

Subir un archivo a la vez no es probablemente lo que la mayoría de los desarrolladores necesitan, ni tampoco usar el elemento **<input>** para seleccionar los archivos a subir. Generalmente todo programador quiere que sus aplicaciones sean lo más intuitiva posibles, y qué mejor manera de lograrlo que combinando técnicas y métodos a los que los usuarios ya están acostumbrados. Aprovechando la API Drag and Drop, vamos a crear una aplicación para subir varios archivos al servidor al mismo tiempo con solo arrastrarlos dentro de un área de la pantalla. El siguiente es el documento con la caja en la que soltar los archivos.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Ajax Level 2</title>
  <link rel="stylesheet" href="ajax.css">
  <script src="ajax.js"></script>
</head>
<body>
  <section id="cajadatos">
    <p>Arrastre y suelte archivos aquí</p>
  </section>
</body>
</html>
```

Listado 21-10: *Definiendo el área donde soltar los archivos a subir*

El código JavaScript para este ejemplo combina dos API y organiza el código con funciones anónimas. Tenemos que tomar los archivos que soltamos dentro del elemento **cajadatos** y listarlos en la pantalla, preparar el formulario con el archivo a enviar, crear una solicitud para subirlo al servidor y actualizar la barra de progreso de cada archivo mientras se están subiendo.

```
var cajadatos;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  cajadatos.addEventListener("dragenter", function(evento) {
    evento.preventDefault();
  });
  cajadatos.addEventListener("dragover", function(evento) {
    evento.preventDefault();
  });
  cajadatos.addEventListener("drop", soltar);
}
function soltar(evento) {
  evento.preventDefault();
  var archivos = evento.dataTransfer.files;
  if (archivos.length) {
    var lista = "";
    for (var f = 0; f < archivos.length; f++) {
      var archivo = archivos[f];
      lista += "<div>Archivo: " + archivo.name;
```

```
      lista += '<br><span><progress value="0" max="100">0%</progress></span>';
      lista += "</div>";
    }
    cajadatos.innerHTML = lista;
    var contador = 0;
    var subir = function() {
      var archivo = archivos[contador];
      var datos = new FormData();
      datos.append("archivo", archivo);
      var url = "procesar.php";
      var solicitud = new XMLHttpRequest();
      var xmlupload = solicitud.upload;
      xmlupload.addEventListener("progress", function(evento) {
        if (evento.lengthComputable) {
          var indice = contador + 1;
          var porcentaje = parseInt(evento.loaded / evento.total * 100);
          var progreso = cajadatos.querySelector("div:nth-child(" +
indice + ") > span > progress");
          progreso.value = porcentaje;
          progreso.innerHTML = porcentaje + "%";
        }
      });
      solicitud.addEventListener("load", function() {
        var indice = contador + 1;
        var elemento = cajadatos.querySelector("div:nth-child(" +
indice + ") > span");
        elemento.innerHTML = "Terminado";
        contador++;
        if (contador < archivos.length) {
          subir();
        }
      });
      solicitud.open("POST", url, true);
      solicitud.send(datos);
    };
    subir();
  }
}
window.addEventListener("load", iniciar);
```

Listado 21-11: *Subiendo archivos uno por uno*

El código del Listado 21-11 no es sencillo, pero será más fácil de estudiar si lo analizamos paso a paso. Como siempre, todo comienza con nuestra función **iniciar()**, a la que se llama tan pronto como se carga el documento. Esta función obtiene una referencia al elemento **cajadatos** donde podremos soltar los archivos y agregar listeners para los tres eventos que controlan la operación de arrastrar y soltar (ver la API Drag and Drop del Capítulo 17). El evento **dragenter** se desencadena cuando los archivos que se están arrastrando entran en el área de la caja, el evento **dragover** se desencadena periódicamente mientras los archivos se encuentran sobre la caja y el evento **drop** se desencadena cuando los archivos se sueltan dentro de la caja. No tenemos que hacer nada con los eventos **dragenter** y **dragover** en este ejemplo, por lo que se cancelan para evitar el comportamiento por defecto del navegador. El único evento al que respondemos es **drop**. La función **soltar()**, declarada para responder a este evento, se ejecuta cada vez que se suelta algo dentro de **cajadatos**.

La primera línea de la función **soltar()** también usa el método **preventDefault()** para hacer lo que queremos con los archivos y no lo que el navegador haría por defecto. Ahora que tenemos control absoluto sobre la situación, es hora de procesar los archivos. Primero, obtenemos los archivos desde el objeto **dataTransfer**. El valor que devuelve es un array de objetos **File** que almacenamos en la variable **files**. Para asegurarnos de que dentro de la caja solo se han soltado archivos y no otros tipos de elementos, controlamos el valor de la propiedad **length**. Si este valor es diferente de **0** o **null**, significa que se han soltado uno o más archivos y podemos proceder.

Es hora de trabajar con los archivos recibidos. Con un bucle **for**, navegamos a través del array **archivos** y creamos una lista de elementos **<div>** que contienen el nombre del archivo y una barra de progreso entre etiquetas **<span>**. Una vez que se termina, el resultado se inserta en el elemento **cajadatos** para mostrarlo en la pantalla.

Parece como que la función **soltar()** hace todo el trabajo, pero dentro de esta función, creamos otra función llamada **subir()** para controlar el proceso de subir los archivos. Por lo tanto, después de mostrar los archivos en pantalla, el siguiente paso es definir esta función y llamarla por cada archivo en la lista.

La función **subir()** se crea usando una función anónima. Dentro de esta función, seleccionamos un archivo desde el array con la variable **contador** como índice. Esta variable se inicializa a **0**, por lo que la primera vez que se llama a la función **subir()**, se selecciona el primer archivo de la lista y se sube.

Cada archivo se sube usando el mismo método que en anteriores ejemplos. Se almacena una referencia al archivo en la variable **archivo**, se crea un objeto **FormData** con el constructor **FormData()** y el archivo se agrega al objeto con el método **append()**.

Esta vez solo respondemos a dos eventos para controlar el proceso: **progress** y **load**. Cada vez que el evento **progress** se desencadena, se llama a una función anónima para actualizar el estado de la barra de progreso del archivo que se está subiendo. El elemento **<progress>** que corresponde al archivo se identifica con el método **querySelector()** y la seudoclase **:nth-child()**. El índice para la seudoclase se calcula usando el valor de la variable **contador**. Esta variable contiene el número de índice del array **archivos**, pero este índice comienza en **0** y el índice que usa la lista de elementos hijos a la que se accede mediante **:nth-child()** comienza por el valor **1**. Para obtener el valor de índice correspondiente y ubicar el elemento **<progress>** correcto, sumamos **1** al valor de **contador**, almacenamos el resultado en la variable **indice** y usamos esta variable como índice.

Cada vez que el proceso anterior finaliza, tenemos que informar de esta situación y continuar con el siguiente archivo del array **archivos**. Para este propósito, en la función anónima ejecutada cuando se desencadena el evento **load**, incrementamos el valor de **contador** en 1, reemplazamos el elemento **<progress>** por el texto "Terminado" y llamamos nuevamente a la función **subir()** si aún quedan archivos por procesar.

La función **subir()** se llama por primera vez al final de la función **soltar()**. Debido a que el valor de **contador** se ha inicializado con el valor **0**, el primer archivo a procesar es el primero del array **archivos**. Cuando el proceso de subir este archivo finaliza, el evento **load** se desencadena y la función anónima a la que se llama para responder a este evento incrementa el valor de **contador** en **1** y nuevamente ejecuta la función **subir()** para procesar el siguiente archivo en el array. Al final, todos los archivos que se sueltan dentro de la caja se suben al servidor, uno por uno.

Capítulo 22
API Web Messaging

22.1 Mensajería

La API Web Messaging permite que las aplicaciones de diferentes orígenes se comuniquen entre sí. El proceso se llama *Cross-Document Messaging*. Las aplicaciones que se ejecutan en diferentes marcos, pestañas o ventanas ahora se pueden comunicar con esta tecnología.

Enviando un mensaje

El procedimiento es sencillo: enviamos un mensaje desde un documento y lo leemos en el documento de destino. La API incluye el siguiente método para enviar mensajes.

postMessage(mensaje, destino**)**—Este método envía un mensaje a otro documento. El atributo **mensaje** es una cadena de caracteres que representa el mensaje a transmitir y el atributo **destino** es el dominio del documento destino (dominio o puerto, como veremos más adelante). El destino se puede declarar como un dominio específico, como cualquier documento con el carácter `*`, o como igual al origen usando el carácter `/`. El método también puede incluir un array de puertos como tercer atributo.

El método de comunicación es asíncrono. La API incluye el siguiente evento para responder a los mensajes que llegan desde otros documentos.

message—Este evento se desencadena cuando se recibe un mensaje.

El evento `message` envía un objeto de tipo `MessageEvent` a la función que responde al mismo, el cual incluye algunas propiedades que devuelven la información sobre del mensaje.

data—Esta propiedad devuelve el contenido del mensaje.

origin—Esta propiedad devuelve el dominio del servidor del documento que ha enviado el mensaje. Este valor se puede utilizar luego para enviar un mensaje de regreso.

source—Esta propiedad devuelve un objeto que identifica a la fuente del mensaje. Este valor se puede usar como una referencia de la fuente para responder al mensaje, como veremos más adelante.

Para crear un ejemplo de esta API, tenemos que considerar que el proceso de comunicación ocurre entre diferentes ventanas (ventanas, marcos o pestañas), por lo que debemos proveer documentos para cada lado de la conversación. Nuestro ejemplo incluye un documento HTML con un iframe (marco) y los códigos JavaScript necesarios para el documento principal y el documento cargado dentro del iframe. El siguiente es el código HTML para el documento principal.

```
<!DOCTYPE html>
<html lang="es">
```

```
<head>
  <meta charset="utf-8">
  <title>Cross Document Messaging</title>
  <link rel="stylesheet" href="messaging.css">
  <script src="messaging.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <label for="nombre">Su nombre: </label>
      <input type="text" name="nombre" id="nombre" required>
      <button type="button" id="boton">Enviar</button>
    </form>
  </section>
  <section id="cajadatos">
    <iframe id="iframe" src="iframe.html" width="500"
height="350"></iframe>
  </section>
</body>
</html>
```

Listado 22-1: *Incluyendo un iframe dentro de un documento para probar la API Web Messaging*

En este ejemplo tenemos dos elementos **<section>**, como en documentos anteriores, pero esta vez el elemento **cajadatos** incluye un elemento **<iframe>** que carga el archivo iframe.html.

Lo básico: el elemento **<iframe>** nos permite insertar un documento dentro de otro documento. El documento para el **<iframe>** lo declara el atributo **src**. Todo lo que se encuentra dentro de un iframe responde como si estuviera localizado en su propia ventana.

Los siguientes son los estilos que necesita nuestro documento para crear las cajas en la pantalla.

```
#cajaformulario {
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos {
  float: left;
  width: 500px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
```

Listado 22-2: *Definiendo las cajas (messaging.css)*

El código JavaScript para el documento principal tiene que tomar los valores del campo **nombre** del formulario y enviarlos al documento dentro del iframe usando el método **postMessage()**.

```
function iniciar() {
  var boton = document.getElementById("boton");
  boton.addEventListener("click", enviar);
}
function enviar() {
  var nombre = document.getElementById("nombre").value;
  var iframe = document.getElementById("iframe");

  iframe.contentWindow.postMessage(nombre, "*");
}
window.addEventListener("load", iniciar);
```

Listado 22-3: *Enviando un mensaje al iframe (messaging.js)*

En el código del Listado 22-3, el mensaje está compuesto por el valor del campo **nombre**. El carácter * se usa como destino para enviar este mensaje a todo documento abierto dentro del iframe, independientemente de su origen.

Lo básico: el método **postMessage()** pertenece al objeto **Window**. Para obtener el objeto **Window** de un iframe desde el documento principal, tenemos que usar la propiedad **contentWindow**.

Cuando se pulsa el botón Enviar en el documento principal, se llama a la función **enviar()** y el valor del campo de entrada se envía al contenido del iframe. Ahora debemos leer este mensaje en el iframe y procesarlo. Para ello vamos a crear un pequeño documento HTML que abriremos en el iframe para mostrar esta información en pantalla.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>iframe</title>
  <script src="iframe.js"></script>
</head>
<body>
  <section>
    <div><b>Mensaje desde la ventana principal:</b></div>
    <div id="cajadatos"></div>
  </section>
</body>
</html>
```

Listado 22-4: *Creando un documento para el iframe (iframe.html)*

Este documento tiene su propia **cajadatos** que podemos usar para mostrar el mensaje en pantalla y el siguiente código JavaScript para procesarlo.

```
function iniciar() {
  window.addEventListener("message", recibir);
}
```

```
function recibir(evento) {
  var cajadatos = document.getElementById("cajadatos");
  cajadatos.innerHTML = "Mensaje desde: " + evento.origin + "<br>";
  cajadatos.innerHTML += "mensaje: " + evento.data;
}
window.addEventListener("load", iniciar);
```

Listado 22-5: *Procesando mensajes en el destino (iframe.js)*

Como ya hemos explicado, para recibir mensajes la API incluye el evento **message** y las propiedades del objeto **MessageEvent**. En el código del Listado 22-5, se declara la función **recibir()** para responder a este evento. La función muestra el contenido del mensaje usando la propiedad **data** e información acerca del documento que ha enviado el mensaje usando el valor de la propiedad **origin**.

Es importante tener siempre presente que este código JavaScript pertenece al documento del iframe, no al documento principal del Listado 22-1. Estos son dos documentos diferentes con sus propios ámbitos y códigos JavaScript; uno se abre en la ventana principal del navegador y el otro se abre dentro del iframe.

Hágalo usted mismo: en este ejemplo tenemos un total de cinco archivos que se deben crear y subir al servidor. Primero, cree un nuevo archivo HTML con el código del Listado 22-1 para el documento principal. Este documento requiere el archivo messaging.css con los estilos del Listado 22-2 y el archivo messaging.js con el código JavaScript del Listado 22-3. El documento del Listado 22-1 contiene un elemento **<iframe>** con el archivo iframe.html como su fuente. Necesita crear este archivo con el código HTML del Listado 22-4 y su correspondiente archivo iframe.js con el código JavaScript del Listado 22-5. Suba todos los archivos a su servidor o servidor local, abra el documento principal en su navegador y envíe su nombre al iframe usando el formulario.

Filtros y origen cruzado

Hasta el momento, nuestro código no ha seguido lo que se considera buena práctica, especialmente considerando cuestiones de seguridad. El código JavaScript del documento principal envía un mensaje al iframe pero no controla qué documento tiene permitido leerlo (cualquier documento dentro del iframe podrá leer el mensaje). Además, el código dentro del iframe no controla el origen y procesa todos los mensajes recibidos. Se deben mejorar ambas partes del proceso de comunicación para prevenir abuso.

En el siguiente ejemplo, vamos a corregir esta situación y demostrar cómo podemos contestar a un mensaje desde el documento destino usando otra propiedad que facilita el objeto **MessageEvent** llamada **source**. Esta es la actualización del documento principal.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Cross Document Messaging</title>
  <link rel="stylesheet" href="messaging.css">
  <script src="messaging.js"></script>
</head>
```

```
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <label for="nombre">Su nombre: </label>
      <input type="text" name="nombre" id="nombre" required>
      <button type="button" id="boton">Enviar</button>
    </form>
  </section>
  <section id="cajadatos">
    <iframe id="iframe" src="http://www.dominio2.com/iframe.html"
width="500" height="350"></iframe>
  </section>
</body>
</html>
```

Listado 22-6: *Comunicándonos con orígenes y destinos específicos*

En el documento del Listado 22-6 no solo declaramos la URL del documento del iframe como hemos hecho anteriormente, sino que además incluimos una ruta que se encuentra en una ubicación diferente a la del documento principal (www.dominio2.com). Para prevenir abusos, tenemos que declarar estas ubicaciones y determinar quién puede leer un mensaje y desde dónde. El código JavaScript para el documento principal ahora considera esta situación.

```
function iniciar() {
  var boton = document.getElementById("boton");
  boton.addEventListener("click", enviar);
  window.addEventListener("message", recibir);
}
function enviar() {
  var nombre = document.getElementById("nombre").value;
  var iframe = document.getElementById("iframe");
  iframe.contentWindow.postMessage(nombre, "http://www.dominio2.com");
}
function recibir(evento) {
  if (evento.origin == "http://www.dominio2.com") {
    document.getElementById("nombre").value = evento.data;
  }
}
window.addEventListener("load", iniciar);
```

Listado 22-7: *Comunicándonos con un origen específico (messaging.js)*

En el método **postMessage()** de la función **enviar()** del Listado 22-7 ahora declaramos el destino para el mensaje (www.dominio2.com). Solo los documentos dentro del iframe y desde ese origen podrán leer los mensajes.

En la función **iniciar()** también agregamos un listener para el evento **message**. El propósito de la función **recibir()** es recibir la respuesta enviada por el documento en el iframe (esto tendrá sentido más adelante).

El código JavaScript para el iframe tiene que procesar mensajes solo desde los orígenes autorizados y enviar una respuesta. La siguiente implementación solo acepta mensajes que provienen de www.dominio1.com.

```
function iniciar() {
  window.addEventListener("message", recibir);
}
function recibir(evento) {
  var cajadatos = document.getElementById("cajadatos");
  if (evento.origin == "http://www.dominio1.com") {
    cajadatos.innerHTML = "Mensaje válido: " + evento.data;
    evento.source.postMessage("Mensaje recibido", evento.origin);
  } else {
    cajadatos.innerHTML = "Origen Invalido";
  }
}
window.addEventListener("load", iniciar);
```

Listado 22-8: *Respondiendo al documento principal (iframe.js)*

El filtro para el origen es tan sencillo como comparar el valor de la propiedad **origin** con el dominio desde el cual queremos leer los mensajes. Una vez que el dominio se detecta como válido, el mensaje se muestra en pantalla y luego se envía una respuesta de vuelta usando el valor de la propiedad **source**. La propiedad **origin** también se usa para declarar esta respuesta, que solo estará disponible para la ventana que ha enviado el mensaje (la función **recibir()** procesa esta respuesta en el Listado 22-7).

Hágalo usted mismo: este ejemplo es un poco complicado. Usamos dos orígenes diferentes, por lo que se necesitan dos dominios separados (o subdominios) para probar los códigos. Reemplace los dominios en los códigos por los suyos, luego suba los códigos para el documento principal a un dominio y los códigos para el iframe al otro dominio, y podrá ver cómo estos dos documentos se comunican entre sí desde orígenes diferentes.

Capítulo 23
API WebSocket

23.1 Web Sockets

La API WebSocket facilita soporte para comunicaciones bidireccionales rápidas y efectivas entre navegadores y servidores. La conexión se establece a través de un socket TCP sin enviar cabeceras HTTP, lo cual reduce el tamaño de los datos transmitidos en cada llamada. La conexión también es persistente, lo que permite a los servidores mantener a los clientes actualizados sin la necesidad de recibir una solicitud previa, y esto significa que no tenemos que llamar al servidor cada vez que necesitamos actualizar los datos. En su lugar, el servidor mismo automáticamente nos envía información sobre la condición actual.

WebSocket se puede confundir con una mejora de Ajax, pero es en realidad una alternativa completamente diferente de comunicación que nos permite construir aplicaciones que responden en tiempo real en una plataforma escalable, como videojuegos multijugador, salas de chat, etc.

La API es sencilla; se incluyen unos pocos métodos y eventos para abrir y cerrar la conexión, y también para enviar y recibir mensajes. Sin embargo, por defecto, ningún servidor ofrece este servicio y la respuesta se tiene que adaptar a nuestras necesidades, por lo que tenemos que instalar nuestro propio servidor WS (servidor WebSocket) para poder establecer comunicación entre el navegador y el servidor que aloja nuestra aplicación.

Servidor WebSocket

Aunque podemos construir nuestro propio servidor WS, existen varios programas para instalar un servidor y prepararlo para procesar solicitudes. Dependiendo de nuestras preferencias, podemos optar por códigos escritos en PHP, Java, Ruby u otros lenguajes. Para el propósito de este capítulo, vamos a usar un servidor PHP. Existen varias versiones disponibles en este lenguaje, pero la que consideramos más fácil de instalar y configurar es un servidor llamado *phpws*, desarrollado por Chris Tanaskoski.

IMPORTANTE: el servidor phpws requiere al menos la versión de PHP 5.3 para funcionar sin problemas, y su servicio de alojamiento (hosting) debe incluir acceso shell para poder comunicarse con su servidor y ejecutar el código PHP (puede consultar con su proveedor de alojamiento para activar el acceso shell si no lo tiene por defecto).

El servidor phpws incluye varios archivos que crean las clases y métodos que necesitamos para ejecutar el servidor. La librería está disponible en https://github.com/Devristo/phpws/, pero para probar nuestros ejemplos, hemos incluido un paquete en nuestro sitio web ya configurado para instalar el servidor WS. El paquete incluye un archivo llamado demo.php que contiene la función `onMessage()` donde se procesan todos los mensajes recibidos por el servidor. Si queremos ofrecer nuestra propia respuesta, tenemos que modificar esta función. La siguiente es un versión de la función que hemos desarrollado para los ejemplos de este capítulo.

```
public function onMessage(IWebSocketConnection $user, IWebSocketMessage $msg){
  $msg = trim($msg->getData());
  switch($msg){
    case 'hola':
      $msgback = WebSocketMessage::create("Hola humano");
      $user->sendMessage($msgback);
      break;
    case 'nombre':
      $msgback = WebSocketMessage::create("No tengo un nombre");
      $user->sendMessage($msgback);
      break;
    case 'edad':
      $msgback = WebSocketMessage::create("Soy viejo");
      $user->sendMessage($msgback);
      break;
    case 'fecha':
      $msgback = WebSocketMessage::create("Hoy es ".date("F j, Y"));
      $user->sendMessage($msgback);
      break;
    case 'hora':
      $msgback = WebSocketMessage::create("La hora es ".date("H:iA"));
      $user->sendMessage($msgback);
      break;
    case 'gracias':
      $msgback = WebSocketMessage::create("No hay problema");
      $user->sendMessage($msgback);
      break;
    case 'adios':
      $msgback = WebSocketMessage::create("Que tengas un buen día");
      $user->sendMessage($msgback);
      break;
    default:
      $msgback = WebSocketMessage::create("No entiendo");
      $user->sendMessage($msgback);
      break;
  }
}
```

Listado 23-1: *Adaptando la función* `onMessage()` *a nuestra aplicación (demo.php)*

Una vez que tenemos todos estos archivos en nuestro servidor, es hora de ejecutar el servidor WS. WebSocket usa una conexión persistente, por lo tanto el servidor WS tiene que funcionar todo el tiempo, recibiendo y enviando mensajes a los usuarios. Para ejecutar el archivo PHP, tenemos que acceder a nuestro servidor mediante una conexión SSH, abrir el directorio donde se encuentra el archivo demo.php e insertar el comando **`php demo.php`**.

Hágalo usted mismo: descargue el archivo ws.zip desde nuestro sitio web, descomprímalo, y suba el directorio ws y todos sus archivo a su servidor. Dentro de este directorio, encontrará el archivo demo.php con la función **`onMessage()`** ya actualizada con el código del Listado 23-1. Conéctese a su servidor con SSH usando su programa preferido (Terminal, Putty, etc.), encuentre el directorio ws que acaba de subir y ejecute el comando **`php demo.php`** para ejecutar el servidor WS.

IMPORTANTE: también puede usar un servidor WS público, como ws://echo.websocket.org/ (para obtener más información, visite http://websocket.org/echo.html). Generalmente estos servidores no incluyen comandos y solo devuelven el mensaje recibido al mismo usuario, pero pueden resultar útiles para probar la API.

Lo básico: SSH es un protocolo de red (Secure Shell) que puede usar para acceder a su servidor y controlarlo de forma remota. Este protocolo le permite trabajar con directorios y archivos en su servidor, y ejecutar programas. Las aplicaciones gratuitas más populares que ofrecen acceso Shell son *Terminal* para ordenadores Apple y *PuTTY* para Windows (disponible en www.chiark.greenend.org.uk/~sgtatham/putty/).

Conectándose al servidor

Con el servidor en marcha, ahora tenemos que programar el código JavaScript que se conectará al mismo. Para este propósito, la API ofrece un objeto llamado `WebSocket` con algunas propiedades, métodos y eventos para configurar la conexión. El siguiente es el constructor de este objeto.

WebSocket(url**)**—Este constructor inicia una conexión entre la aplicación y el servidor WS apuntado por el atributo **url**. El constructor devuelve un objeto `WebSocket` referenciando la conexión. Se puede especificar un segundo atributo para facilitar un array con subprotocolos de comunicación.

El constructor inicia la conexión, por lo que solo necesitamos dos métodos para trabajar con la misma.

send(datos**)**—Este método envía un mensaje al servidor WS. El atributo **datos** es una cadena de caracteres con la información a transmitir.

close()—Este método cierra la conexión.

Unas pocas propiedades informan acerca de la configuración y el estado de la conexión.

url—Esta propiedad devuelve la URL a que está conectada la aplicación.

protocol—Esta propiedad devuelve el subprotocolo utilizado.

readyState—Esta propiedad devuelve un número que representa el estado de la conexión: 0 significa que la conexión aún no se ha establecido, 1 significa que la conexión está abierta, 2 significa que la conexión se está cerrando y 3 significa que la conexión se ha cerrado.

bufferedAmount—Esta propiedad informa de la cantidad de datos solicitados por la conexión pero que todavía no se han enviado al servidor. El valor devuelto nos ayuda a regular la cantidad de datos enviados y la frecuencia de las solicitudes para no saturar el servidor.

La API también incluye los siguientes eventos para conocer el estado de la conexión y responder a los mensajes enviados por el servidor.

open—Este evento se desencadena cuando se inicia la conexión.

message—Este evento se desencadena cuando hay un mensaje del servidor disponible.

error—Este evento se desencadena cuando ocurre un error.

close—Este evento se desencadena cuando se cierra la conexión.

El archivo demo.php que hemos preparado para estos ejemplos contiene un método llamado **`onMessage()`** que procesa una pequeña lista de comandos y envía de regreso la respuesta adecuada (ver Listado 23-1). Para probar este código, vamos a usar un formulario con el que insertar estos comandos y enviarlos al servidor.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>WebSocket</title>
  <link rel="stylesheet" href="websocket.css">
  <script src="websocket.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <label for="comando">Comando: </label><br>
      <input type="text" name="comando" id="comando"><br>
      <button type="button" id="boton">Enviar</button>
    </form>
  </section>
  <section id="cajadatos"></section>
</body>
</html>
```

Listado 23-2: *Creando un documento para insertar comandos*

También necesitaremos un archivo CSS con los siguientes estilos para diseñar las cajas en la página.

```
#cajaformulario {
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos {
  float: left;
  width: 500px;
  height: 350px;
  overflow: auto;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
```

Listado 23-3: *Definiendo los estilos para las cajas*

Como siempre, el código JavaScript es responsable de todo el proceso. El siguiente ejemplo establece una comunicación sencilla con el servidor para probar esta API.

```
var cajadatos, socket;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  var boton = document.getElementById("boton");
  boton.addEventListener("click", enviar);

  socket = new WebSocket("ws://SU_DIRECCION_IP:12345/ws/demo.php");
  socket.addEventListener("message", recibido);
}
function recibido(evento) {
  var lista = cajadatos.innerHTML;
  cajadatos.innerHTML = "Recibido: " + evento.data + "<br>" + lista;
}
function enviar() {
  var comando = document.getElementById("comando").value;
  socket.send(comando);
}
window.addEventListener("load", iniciar);
```

Listado 23-4: *Enviando mensajes al servidor*

IMPORTANTE: reemplace el texto **SU_DIRECCION_IP** por la IP de su servidor. También puede especificar su dominio, pero usando la IP evita el proceso de traducción DNS. Siempre debería usar esta técnica para acceder a su aplicación y evitar el tiempo que pierde la red en traducir el domino a la dirección IP. Además, su servidor tiene que tener el puerto 12345 abierto para utilizar los códigos JavaScript y el servidor WS provistos en este capítulo. Si no está seguro de cómo abrir este puerto, consulte con su proveedor de alojamiento.

En la función **iniciar()** del Listado 23-4, el objeto **WebSocket** se construye y almacena en la variable **socket**. El atributo **url** declarado en el constructor apunta a la ubicación del archivo demo.php en nuestro servidor. Esta URL incluye el puerto de conexión. Normalmente, el servidor se especifica mediante su dirección IP y el valor del puerto se declara como 12345, pero esto depende de nuestras necesidades, la configuración del servidor, los puertos disponibles, la ubicación del archivo en nuestro servidor, etc.

Después de obtener el objeto **WebSocket**, agregamos un listener para el evento **message**. Cada vez que el servidor WS envía un mensaje al navegador, se desencadena el evento **message** y se llama a la función **recibido()** para responder. Como con anteriores API, el objeto enviado por este evento a la función incluye la propiedad **data** que contiene el mensaje. En la función **recibido()** leemos esta propiedad para mostrar el mensaje en la pantalla.

Se incluye la función **enviar()** para enviar mensajes al servidor. El valor del elemento **<input>** lo toma esta función y lo envía al servidor WS usando el método **send()**.

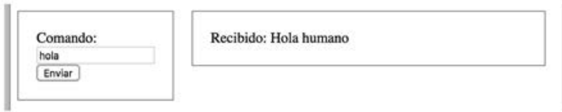

Figura 23-1: *Aplicación comunicándose con el servidor WS*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento Listado 23-2, un archivo CSS llamado websocket.css con el código del Listado 23-3 y un archivo JavaScript llamado websocket.js con el código del Listado 23-4. Abra el documento de su navegador, inserte el comando **hola** en el campo de entrada y pulse el botón Enviar. Debería ver un mensaje en la caja de la derecha con la respuesta del servidor, tal como ilustra la Figura 23-1 (su servidor WS se tiene que instalar y ejecutar, según hemos explicado en la sección previa de este capítulo).

IMPORTANTE: la función **`onMessage()`** que hemos preparado para estos ejemplos compara el mensaje recibido con una lista de comandos predefinidos (vea el Listado 23-1). Los comandos disponibles son **hola**, **nombre**, **edad**, **fecha**, **hora**, **gracias** y **adios**.

El último ejemplo ilustra cómo trabaja el proceso de comunicación de esta API. La conexión la inicia el constructor **`WebSocket()`**, el método **`send()`** envía al servidor todos los mensajes que queremos procesar y el evento **`message`** informa a la aplicación cuando se reciben nuevos mensajes desde el servidor. Sin embargo, no hemos cerrado la conexión, no hemos controlado errores o detectado cuándo la conexión estaba lista para trabajar. El siguiente ejemplo responde a todos los eventos que facilita la API para informar al usuario acerca del estado de la conexión en cada paso del proceso.

```
var cajadatos, socket;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  var boton = document.getElementById("boton");
  boton.addEventListener("click", enviar);
  socket = new WebSocket("ws://SU_DIRECCION_IP:12345/ws/demo.php");
  socket.addEventListener("open", abierta);
  socket.addEventListener("message", recibido);
  socket.addEventListener("close", cerrada);
  socket.addEventListener("error", mostrarerror);
}
function abierta() {
  cajadatos.innerHTML = "CONEXION ABIERTA<br>";
  cajadatos.innerHTML += "Estado: " + socket.readyState;
}
function recibido(evento) {
  var lista = cajadatos.innerHTML;
  cajadatos.innerHTML = "Recibido: " + evento.data + "<br>" + lista;
}
function cerrada() {
  var lista = cajadatos.innerHTML;
  cajadatos.innerHTML = "CONEXION CERRADA<br>" + lista;
  var boton = document.getElementById("boton");
  boton.disabled = true;
}
function mostrarerror() {
  var lista = cajadatos.innerHTML;
  cajadatos.innerHTML = "ERROR<br>" + lista;
}
```

```
function enviar() {
  var comando = document.getElementById("comando").value;
  if (comando == "cerrar") {
    socket.close();
  } else {
    socket.send(comando);
  }
}
window.addEventListener("load", iniciar);
```

Listado 23-5: *Informando al usuario acerca del estado de la conexión*

En el código del Listado 23-5 se han introducido algunas mejoras con respecto al ejemplo anterior. En este ejemplo hemos agregado listeners para todos los eventos disponibles en el objeto **`WebSocket`** y se han creado las funciones correspondientes para responder a estos eventos. También mostramos el estado cuando se inicia la conexión usando el valor de la propiedad **`readyState`**, cerramos la conexión usando el método **`close()`** cuando el comando "cerrar" se envía desde el formulario, y deshabilitamos el botón Enviar cuando se cierra la conexión (**`boton.disabled = true`**).

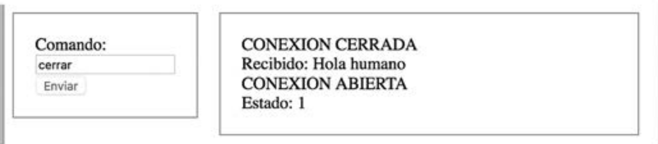

Figura 23-2: *Conexión cerrada*

Hágalo usted mismo: este último ejemplo requiere el documento HTML y los estilos de los Listados 23-2 y 23-3. Abra el documento en su navegador, inserte un comando en el formulario y haga clic en el botón Enviar. Debería recibir respuestas desde el servidor según el comando insertado (**`hola`**, **`nombre`**, **`edad`**, **`fecha`**, **`hora`**, **`gracias`** o **`adios`**). Inserte el comando "cerrar" para cerrar la conexión.

IMPORTANTE: recuerde que su servidor WS tiene que funcionar constantemente para poder procesar las solicitudes. Como hemos mencionado antes, si lo desea, puede probar estos ejemplos con un servidor WS público, como ws://echo.websocket.org/. Este servidor enviará de regreso el mismo texto insertado en el formulario.

Capítulo 24
API WebRTC

24.1 Paradigmas Web

WebRTC significa Web Real-Time Communication (comunicaciones web en tiempo real). No se trata solo de un sistema de comunicaciones, sino más bien de un nuevo sistema de comunicación para la Web. Esta API permite a los desarrolladores crear aplicaciones que conectan a los usuarios entre sí sin intermediarios. Las aplicaciones que implementan WebRTC pueden transmitir vídeo, audio y datos directamente de un usuario a otro.

Este es un cambio significativo de paradigmas. En el paradigma actual, los usuarios solo pueden compartir información en Internet a través de un servidor. Los servidores son como repositorios gigantescos de contenido accesible a través de un dominio o una IP. Los usuarios se deben conectar a estos servidores, descargar o subir información y esperar que otros usuarios hagan lo mismo desde el otro lado. Si queremos compartir una fotografía con un amigo, por ejemplo, tenemos que subirla al servidor y esperar a que nuestro amigo se conecte al mismo servidor y descargue la fotografía en su ordenador. El proceso se ilustra en la Figura 24-1.

Figura 24-1: *Paradigma actual*

No existe forma de que el Usuario 1 envíe información al Usuario 2 sin usar un servidor. Cada proceso requiere un servidor para almacenar la información enviada por un usuario y ofrecerla (servirla) al otro. Fuera de la Web, existen múltiples aplicaciones que conectan a los usuarios directamente, permitiéndoles enviar mensajes instantáneas o hacer videollamadas, pero los navegadores no eran capaces de hacerlo hasta que se implementó la API WebRTC.

WebRTC nos ofrece un nuevo paradigma para la Web. La API provee la tecnología que nos permite crear aplicaciones de conexión directa. El proceso usa un servidor de señalización para establecer la conexión, pero la información se intercambia entre los navegadores sin más intervención, tal como muestra la Figura 24-2.

Figura 24-2: *Nuevo paradigma para la Web*

Los servidores aún son necesarios, pero ya no son los proveedores de contenido. El servidor ahora envía señales para iniciar la conexión, pero el contenido se transmite directamente entre los ordenadores de los usuarios. Con esta tecnología, la transmisión de audio y vídeo entre usuarios, las videollamadas, la mensajería instantánea y el intercambio de datos ahora son funciones disponibles para aplicaciones que se ejecutan en el navegador.

Servidores ICE

En el paradigma actual los servidores son de acceso público. Estos servidores tienen una IP asignada para identificarlos y normalmente se crea un dominio para representar esa IP, lo que facilita la tarea de determinar su ubicación para los usuarios. Los servidores no son solo fáciles de acceder, sino que además son siempre accesibles, desde cualquier parte del mundo. Si un servidor cambia su IP, su dominio se traduce a la nueva IP casi de inmediato (la información tarda unas horas en propagarse por la red, pero el cambio es instantáneo). Los ordenadores de los usuarios, en cambio, no tienen una IP única, cada usuario recibe una IP privada que luego se traduce a una IP pública mediante un sistema llamado *NAT* (Network Address Translator). Este sistema establece rutas hacia los ordenadores de los usuarios que a veces son complejas de seguir y varían en cada caso. Como si esto no fuera lo suficientemente complicado, los ordenadores de los usuarios también se encuentran ocultos detrás de firewalls, tienen diferentes puertos asignados para transferir datos, e incluso se vuelven inaccesibles sin advertir al sistema. Conectar un usuario con otro requiere información que los navegadores son incapaces de facilitar y, por lo tanto, necesitan ayuda.

Considerando esta situación, la API WebRTC se desarrolló para trabajar junto con servidores que obtienen y devuelven la información necesaria para acceder al ordenador del usuario. Estos servidores trabajan bajo una estructura llamada *ICE* (Interactive Connectivity Establishment) para encontrar la mejor manera de conectar a un usuario con otro. ICE es el nombre de un proceso que obtiene información a través de diferentes servidores y sistemas. La Figura 24-3 muestra cómo se configura la estructura final.

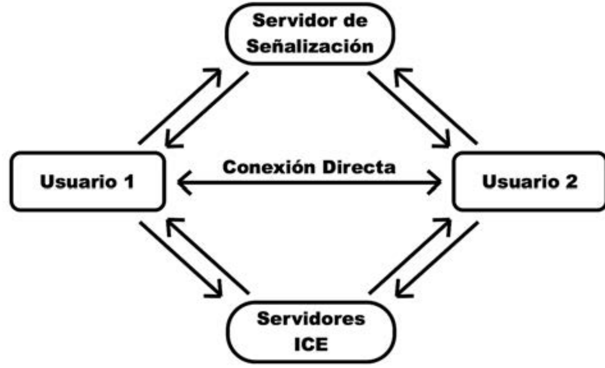

Figura 24-3: *Servidores ICE*

Los servidores ICE son servidores STUN o TURN. Un servidor STUN descubre y devuelve la IP pública del ordenador del usuario y también información sobre cómo se ha configurado la NAT de ese ordenador, mientras que un servidor TURN ofrece una conexión usando una IP en la nube cuando las otras alternativas no están disponibles. Los sistemas a ambos lados de la comunicación deciden qué conexión es la más eficaz y proceden a través de esa ruta.

La estructura descrita en la Figura 24-3 trabaja de la siguiente manera: los navegadores de los Usuarios 1 y 2 acceden a los servidores ICE para obtener información que describe cómo se ve cada ordenador en la red. Ambas aplicaciones acceden al servidor de señalización y envían sus descripciones al otro ordenador. Una vez que se recibe esta información y ambos lados concuerdan la ruta a seguir, se establece la conexión y los usuarios se conectan entre sí sin requerir una nueva intervención de los servidores (a menos que se desconecten y la conexión se tenga que establecer de nuevo).

Conexión

El primer paso que debemos realizar es crear la conexión y facilitar al navegador información sobre la misma, los datos a compartir y los servidores ICE que queremos usar. Esto se realiza a través de propiedades, métodos y eventos que facilita el objeto `RTCPeerConnection`. La API incluye el siguiente constructor para obtener este objeto.

RTCPeerConnection(configuración**)**—Este constructor devuelve un objeto `RTCPeerConnection`. El objeto se usa para facilitar al navegador la información que necesita para establecer la conexión. El atributo **configuración** es un objeto que especifica información de los servidores ICE que vamos a utilizar.

Los usuarios tienen que contar con la posibilidad de iniciar y finalizar una conexión. Los objetos `RTCPeerConnection` incluyen el siguiente método para terminar la conexión.

close()—Este método cambia el estado del objeto `RTCPeerConnection` a `closed` (cerrado), termina tanto la transmisión como los procesos ICE y cierra la conexión.

Candidato ICE

Cuando el proceso ICE encuentra una manera de comunicarse con un ordenador, devuelve información llamada *ICE Candidate*. Los candidatos se gestionan mediante un objeto de tipo `RTCIceCandidate`. La API ofrece el siguiente constructor para obtener estos objetos.

RTCIceCandidate(información**)**—Este constructor devuelve un objeto `RTCIceCandidate`. El atributo **información** facilita información para inicializar el objeto (esta es la información que envía la conexión remota).

La API dispone de los siguientes métodos y propiedades para controlar y agregar el candidato a la conexión.

iceState—Esta propiedad devuelve el estado del proceso ICE para la conexión actual.

addIceCandidate(candidato**)**—Este método agrega un candidato ICE remoto a la conexión. El atributo **candidato** es el objeto que devuelve el constructor `RTCIceCandidate()`.

Ofertas y respuestas

La comunicación entre los usuarios se inicia por medio de dos procesos llamados *oferta* (offer) y *respuesta* (answer). Cuando una aplicación quiere iniciar una conexión, envía una oferta a la

aplicación del otro lado a través de un servidor de señalización. Si la oferta es aceptada, la segunda aplicación envía de regreso una respuesta. Los siguientes son los métodos que facilita la API para generar ofertas y respuestas.

createOffer(éxito, error**)**—Este método se utiliza para crear una oferta. El método genera un blob que contiene la descripción del ordenador local (llamada *Session Description*). La descripción contiene las transmisiones de medios, los codificadores negociados para la sesión, los candidatos ICE, así como una propiedad que describe su tipo (en este caso `offer`). La descripción de la sesión obtenida por este método se envía a la función especificada por el atributo **éxito** para ser procesada. Esta función es responsable de enviar la oferta al ordenador remoto.

createAnswer(éxito, error**)**—Este método se usa para crear una respuesta. El método genera un blob que contiene la descripción del ordenador local (llamada *Session Description*). La descripción contiene las transmisiones de medios, los codificadores negociados para la sesión, los candidatos ICE y también una propiedad que describe su tipo (en este caso `answer`). La descripción de la sesión obtenida por este método se envía a la función especificada por el atributo **éxito** para su procesamiento. Esta función es responsable de enviar la respuesta al ordenador remoto.

Descripción de la sesión

Como acabamos de discutir, la descripción de cada ordenador, incluidas las transmisiones de medios, los codificadores negociados para la sesión y los candidatos ICE, se llama *Session Description*. Estas descripciones se envían de un usuario al otro a través del servidor de señalización y luego se asignan a la conexión con los siguientes métodos.

setLocalDescription(descripción, éxito, error**)**—Este método facilita la descripción del ordenador local a la conexión. El atributo **descripción** es el objeto que devuelve los métodos `createOffer()` y `createAnswer()`. Los atributos **éxito** y **error** son funciones a las que se llamará en caso de éxito o fracaso.

setRemoteDescription(descripción, éxito, error**)**—Este método facilita la descripción del ordenador remoto a la conexión. El atributo **descripción** es un objeto que devuelve el constructor `RTCSessionDescription()` y la información recibida desde el ordenador remoto. Los atributos **éxito** y **error** son funciones a las que se llamará en caso de éxito o fracaso.

Transmisiones de medios

Este tipo de conexiones se crean para compartir transmisiones de medios y datos. El proceso de transmitir datos implica la creación de canales de datos, como veremos más adelante, pero para agregar o eliminar transmisiones de medios a una conexión, solo tenemos que aplicar los siguientes métodos.

addStream(transmisión**)**—Este método agrega una transmisión de medio a la conexión. El atributo **transmisión** es una referencia a la transmisión de medio (que devuelven los métodos como `getUserMedia()`, por ejemplo).

removeStream(transmisión**)**—Este método elimina una transmisión de medio de una conexión. El atributo **transmisión** es una referencia a la transmisión de medio (que devuelven los métodos como `getUserMedia()`, por ejemplo).

Eventos

El proceso de establecer la conexión y obtener información desde uno y otro ordenador es asíncrono. Una vez que se crea el objeto `RTCPeerConnection`, tenemos que responder a eventos para procesar los resultados. La siguiente es la lista de eventos disponibles.

negotiationneeded—Este evento se desencadena cuando se necesita una nueva sesión de negociaciones (por ejemplo, se debe enviar una nueva oferta).

icecandidate—Este evento se desencadena cuando hay disponible un nuevo candidato ICE.

statechange—Este evento se desencadena cuando cambia el estado de la conexión.

addstream—Este evento se desencadena cuando el ordenador remoto agrega una transmisión de medio.

removestream—Este evento se desencadena cuando el ordenador remoto elimina una transmisión de medio.

gatheringchange—Este evento se desencadena cuando cambia el estado del proceso ICE.

icechange—Este evento se desencadena cuando cambia el estado de ICE (por ejemplo, un nuevo servidor fue declarado).

datachannel—Este evento se desencadena cuando el ordenador remoto agrega un nuevo canal de datos.

24.2 Configuración

Como hemos explicado en la introducción de este capítulo, la estructura necesaria para generar una conexión de este tipo no solo involucra el código JavaScript de cada lado sino además servidores que establezcan y coordinen la conexión. La API deja la configuración del proceso, la selección de las tecnologías utilizadas, así como los tipos de servidores y redes a utilizar en manos del desarrollador, por lo que hay muchas cosas que hacer además de programar la aplicación. Una de las más importantes es configurar el servidor de señalización.

IMPORTANTE: los ejemplos de este capítulo se basan en un procedimiento que consideramos apropiado en estas circunstancias, pero el modo de identificar a los usuarios puede cambiar completamente en otros contextos. Usted deberá decidir si este procedimiento es adecuado para su aplicación o no.

Configurando el servidor de señalización

Un servidor de señalización no es lo mismo que un servidor de contenido. Los servidores de señalización tienen que establecer conexiones persistentes para poder informar a la aplicación

cuándo se recibe una oferta o una respuesta. La Figura 24-3 ilustra este proceso. Cuando el Usuario 1 quiere conectarse al Usuario 2, la aplicación del ordenador del Usuario 1 tiene que enviar una oferta al servidor de señalización solicitando la conexión y el servidor tiene que poder informar al Usuario 2 de que se ha recibido una solicitud de conexión. Este proceso solo es posible si ya se ha establecido una conexión persistente entre el servidor y ambos usuarios. Por lo tanto, un servidor de señalización no solo tiene el propósito de recibir y enviar señales (ofertas y respuestas), sino que también tiene que mantener a los usuarios informados estableciendo una conexión permanente antes de que se configure la conexión entre los usuarios (por ejemplo, un sistema de llamadas de vídeo no sería posible si a los usuarios no se les notificara de una llamada entrante).

El trabajo de un servidor de señalización no finaliza aquí. También debe controlar quién está autorizado a crear una conexión y quién puede conectarse con quién. WebRTC no ofrece un método estándar para hacerlo; todo queda en manos del desarrollador. La buena noticia es que existen varias opciones disponibles. Ya hemos estudiado cómo crear conexiones persistentes en el Capítulo 23 mediante WebSockets. La API WebSocket es una alternativa, pero no la única. Google también ofrece la API Google Channel: una API desarrollada para crear conexiones persistentes entre aplicaciones y los servidores de Google. Además ya hay disponibles de forma gratuita servidores de código abierto que implementan una tecnología llamada *SIP* (Session Initiation Protocol).

Para nuestro ejemplo, hemos decidido implementar el mismo servidor WebSocket del Capítulo 23. La función **onMessage()** del archivo demo.php se tiene que modificar para recibir y enviar señales de regreso hacia la conexión adecuada.

```
public function onMessage(IWebSocketConnection $user, IWebSocketMessage
$msg){
  $thisuser = $user->getId();
  $msg = trim($msg->getData());
  $msgback = WebSocketMessage::create($msg);
  foreach($this->server->getConnections() as $user){
    if($user->getId() != $thisuser){
      $user->sendMessage($msgback);
    }
  }
}
```

Listado 24-1: *Respondiendo a ofertas y respuestas desde el servidor WebSocket (demo.php)*

El servidor de nuestro ejemplo no realiza ningún control, solo lleva a cabo la tarea más básica que es ayudar a establecer la conexión: toma los mensajes recibidos desde un usuario y los envía al otro. Esto no es útil en una aplicación profesional, pero es suficiente para probar nuestra pequeña aplicación y entender cómo funciona el proceso.

Hágalo usted mismo: al igual que en el Capítulo 23, hemos preparado un archivo con el servidor WebSocket listo para ser instalado y con el archivo demo.php ya adaptado a nuestra aplicación. Visite nuestro sitio web, descargue el paquete webrtc.zip y suba el directorio ws que se encuentra dentro de este paquete a su servidor. Acceda a su servidor con SSH usando un programa como PuTTY (disponible en www.chiark.greenend.org.uk/~sgtatham/putty/), abra el directorio ws, y escriba el comando **php demo.php** para ejecutar el servidor WebSocket. Para obtener más información, vea el Capítulo 23.

Configurando los servidores ICE

Los servidores ICE se especifican durante la construcción de la conexión. La API utiliza una propiedad para declarar un array que incluye la configuración de los servidores ICE para el constructor `RTCPeerConnection()`.

iceServers—Esta propiedad se usa para especificar los servidores STUN y TURN disponibles para que los use el proceso ICE.

El valor de esta propiedad se define como un array de objetos que contiene las siguientes propiedades.

urls—Esta propiedad declara la URL del servidor STUN o TURN.

credential—Esta propiedad se usa para definir una credencial para un servidor TURN.

La sintaxis para introducir esta información es la siguiente: `{"iceServers": [{"urls": "stun: stun.dominio.com:12345"}]}`, donde `stun.dominio.com` es el dominio del servidor STUN y `12345` es el puerto en el que está disponible el servidor.

IMPORTANTE: tenemos que facilitar nuestro propio servidor ICE para trabajar con nuestra aplicación. La creación y configuración de servidores STUN y TURN va más allá del propósito de este libro. En los ejemplos de este capítulo vamos a trabajar con el servidor STUN de Google. Para obtener más información sobre servidores ICE, visite nuestro sitio web y siga los enlaces de este capítulo.

24.3 Implementando WebRTC

El uso más común de este tipo de conexiones es el de realizar llamadas de vídeo, por lo que vamos a crear una aplicación que conecta a un usuario con otro para realizar una llamada. El documento de este ejemplo tiene que incluir dos elementos `<video>` con los que mostrar el vídeo de la cámara local y el vídeo de la cámara remota (la persona a la que llamamos).

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API WebRTC</title>
  <style>
    #local, #remoto {
      float: left;
      margin: 5px;
      padding: 5px;
      border: 1px solid #000000;
    }
  </style>
  <script src="webrtc.js"></script>
</head>
<body>
```

```
  <section id="local">
    <video id="localmedio" width="150" height="150"></video>
    <br><input type="button" id="botonllamar" value="Llamar">
  </section>
  <section id="remoto">
    <video id="remotomedio" width="500" height="500"></video>
  </section>
</body>
</html>
```

Listado 24-2: *Creando un documento para hacer llamadas de video*

El código JavaScript tiene que tomar las transmisiones de vídeo y audio creadas por la cámara y el micrófono en el ordenador local, asignarlas al elemento **<video>** con el nombre **localmedio** y enviar la transmisión al otro usuario. Cuando una transmisión de medio llega desde el otro lado de la línea, el código tiene que asignar la transmisión remota al segundo elemento **<video>** identificado con el nombre **remotomedio**, para establecer la comunicación.

Para ejecutar estas tareas, el código debe configurar la conexión, comunicarse con los servidores ICE, obtener la descripción de la sesión, enviar la oferta y la respuesta, y agregar las transmisiones de medios locales y remotos a la conexión. Pero vamos a hacerlo paso a paso. Primero, necesitamos crear la conexión persistente con el servidor de señalización (en este caso, un servidor WebSocket) y luego capturar las transmisiones de vídeo y audio desde la cámara y el micrófono locales.

```
var usuario, socket;
function iniciar() {
  var botonllamar = document.getElementById("botonllamar");
  botonllamar.addEventListener("click", hacerllamada);

  socket = new WebSocket("ws://SU_DIRECCION_IP:12345/ws/demo.php");
  socket.addEventListener("message", recibido);

  var promesa = navigator.mediaDevices.getUserMedia({video: true});
  promesa.then(prepararcamara);
  promesa.catch(mostrarerror);
}
```

Listado 24-3: *Conectando con el servidor WebSocket y accediendo a la transmisión de medios*

En la función **iniciar()** del Listado 24-3, comenzamos declarando la función **hacerllamada()** para responder al evento **click**. Cada vez que se pulsa el botón Llamar, se ejecuta esta función para enviar una oferta e iniciar la llamada. A continuación, la aplicación crea una conexión persistente con el servidor WebSocket y agrega un listener para el evento **message** para recibir los mensajes que proceden del servidor. Esto es útil en ambos lados de la línea: la persona que recibe la solicitud sabrá que está recibiendo una llamada y la persona que llama estará lista para recibir la respuesta.

Finalmente, las transmisiones de medio desde la cámara y el micrófono local se capturan con el método **getUserMedia()** (ver Capítulo 9). En caso de éxito, llamamos a la función **prepararcamara()**, y en caso de que ocurra un error, se ejecuta la función **mostrarerror()** (por ejemplo, cuando el usuario no autoriza el acceso a la cámara). La siguiente es la implementación de estas funciones.

```
function prepararcamara(transmision) {
  var video = document.getElementById("localmedio");
  video.srcObject = transmision;
  video.play();

  var servidores = {"iceServers": [{"urls": "stun:
stun.l.google.com:19302"}]};
  usuario = new RTCPeerConnection(servidores);
  usuario.addStream(transmision);
  usuario.addEventListener("addstream", prepararremoto);
  usuario.addEventListener("icecandidate", prepararice);
}
function mostrarerror() {
  console.log("Error");
}
```

Listado 24-4: *Iniciando la conexión*

Se va a llamar a la función **mostrarerror()** con varios métodos, por lo que la usamos solo para mostrar un mensaje de error en la consola, pero la función **prepararcamara()** tiene más trabajo que hacer. En esta función tenemos que configurar la conexión y asignar las transmisiones de medios al elemento **<video>**. Las transmisiones de vídeo y audio capturadas por el método **getUserMedia()** se asignan al elemento **<video>** correspondiente (el pequeño vídeo del lado izquierdo de la pantalla) y luego se reproducen con el método **play()**. La conexión se inicia a continuación declarando los servidores ICE disponibles y creando el objeto **RTCPeerConnection** con esta información. El servidor STUN de Google se declara como nuestro servidor ICE para este ejemplo.

El objeto **RTCPeerConnection** facilita al navegador la información necesaria para establecer la conexión. Parte de esta información, como la transmisión del medio local, está disponible de inmediato, pero el resto del proceso es asíncrono, por lo que tenemos que responder a eventos para obtener la información restante cuando esté disponible. Por esta razón, después de agregar la transmisión del medio local al objeto con el método **addStream()**, agregamos listeners para los eventos **addstream** y **icecandidate**. El evento **addstream** se desencadenará cuando el ordenador remoto agregue una transmisión remota, y el evento **icecandidate** se desencadenará cuando el ordenador local determine un candidato ICE.

Para que todo esto ocurra, primero se tiene que establecer la conexión. La siguiente es la función que usamos para crear la oferta e iniciar la llamada.

```
function hacerllamada() {
  usuario.createOffer(preparardescripcion, mostrarerror);
}
```

Listado 24-5: *Obteniendo la descripción de la sesión*

Cuando el vídeo y el audio de la cámara y el micrófono local se están reproduciendo en la pantalla en ambos lados de la comunicación, podemos para hacer la llamada. Este proceso comienza cuando uno de los usuarios pulsa el botón Llamar y se ejecuta la función **hacerllamada()**. Esta función genera una descripción de la sesión de tipo offer y llama a la función **preparardescripcion()** con esta información si la operación se realiza

correctamente. La función **preparardescripcion()** recibe la descripción de la sesión, establece esta información como descripción local y la envía al ordenador remoto para iniciar la conexión.

```
function preparardescripcion(descripcion) {
  usuario.setLocalDescription(descripcion, function() {
    enviarmensaje(descripcion);
  });
}
```

***Listado 24-6:** Enviando una oferta a un ordenador remoto*

El método **setLocalDescription()** se usa en la función del Listado 24-6 para facilitar al navegador información que describa el ordenador local. En caso de éxito, esta información se envía luego al ordenador remoto con la función **enviarmensaje()**. Esta es la función a cargo de enviar mensajes al servidor WebSocket para poder establecer la conexión.

```
function enviarmensaje(mensaje) {
  var mns = JSON.stringify(mensaje);
  socket.send(mns);
}
```

***Listado 24-7:** Enviando señales al ordenador remoto*

Antes de enviar objetos JavaScript al servidor tenemos que convertirlos a código JSON. En la función del Listado 24-7, lo hacemos implementando el método **JSON.stringify()**.

Lo básico: JSON (JavaScript Object Notation) es un formato de datos desarrollado específicamente para compartir datos en línea. Un valor JSON es texto con un formato específico que se puede enviar como una cadena de texto regular, pero que se transforma en objetos útiles por casi todos los lenguajes de programación disponibles. JavaScript incluye dos métodos para trabajar con JSON: **JSON.stringify()** para convertir objetos JavaScript a código JSON y **JSON.parse()** para convertir código JSON en objetos JavaScript.

La función **enviarmensaje()** se encarga del proceso de señalización. Cada vez que se realiza una oferta, se envía una respuesta o los ordenadores comparten candidatos ICE, la información se transmite al servidor WebSocket a través de mensajes enviados por esta función. El formato de estos mensajes y el modo en que estas señales se envían y procesan lo decide el desarrollador. Para nuestra aplicación, vamos a usar la propiedad **type** enviada por la descripción de la sesión para confirmar el tipo de mensaje recibido. La siguiente es la función que recibe y procesa las señales.

```
function recibido(evento) {
  var mns = JSON.parse(evento.data);
  switch (mns.type) {
    case "offer":
```

```
      usuario.setRemoteDescription(new RTCSessionDescription(mns),
function() {
          usuario.createAnswer(preparardescripcion, mostrarerror);
        });
      break;
    case "answer":
      usuario.setRemoteDescription(new RTCSessionDescription(mns));
      break;
    case "candidate":
      var candidato = new RTCIceCandidate(mns.candidate);
      usuario.addIceCandidate(candidato);
  }
}
```

Listado 24-8: *Procesando las señales*

Se llama a la función **recibido()** del Listado 24-8 cada vez que se desencadena el evento **message** del objeto **WebSocket**. Esto significa que cada vez que el servidor WebSocket envía un mensaje a la aplicación, esta función se ejecuta. En la misma, controlamos el valor de la propiedad **type** en cada mensaje y procedemos de acuerdo a cada caso. Si el valor de la propiedad **type** es "offer" (oferta), significa que la aplicación ha recibido una oferta desde otro ordenador. En este caso, tenemos que establecer la descripción de la sesión para el ordenador remoto y, en caso de éxito, enviar una respuesta con el método **createAnswer()**. Por otro lado, si el mensaje de señalización es de tipo "answer" (respuesta), lo cual significa que la llamada se ha aceptado, declaramos la descripción del ordenador remoto y se establece la conexión. El último caso de la instrucción **switch** comprueba si el mensaje de señalización es del tipo "candidate". Si es así, el candidato ICE enviado por el ordenador remoto se agrega al objeto **RTCPeerConnection** con el método **addIceCandidate()**.

IMPORTANTE: el procedimiento que acabamos de describir es el que decidimos usar para este ejemplo. La API WebRTC no define ningún procedimiento estándar para procesar mensajes o incluso enviarlos (el uso del servidor WebSocket ha sido también de nuestra elección). Usted debe decidir si este procedimiento es el adecuado para su aplicación o no.

Cuando se acepta la oferta y se envía la respuesta, ambos ordenadores comparten las transmisiones de medios y la información acerca de los servidores ICE que se van a usar para establecer la conexión. Este proceso dispara los eventos **addstream** y **icecandidate**, y ejecuta las funciones **prepararremoto()** y **prepararice()** a ambos lados de la comunicación.

```
function prepararremoto(evento) {
  var video = document.getElementById("remotomedio");
  video.srcObject = evento.stream;
  video.play();
}
function prepararice(evento) {
  if (evento.candidate) {
    var mensaje = {
      type: "candidate",
```

```
      candidate: evento.candidate,
    };
    enviarmensaje(mensaje);
  }
}
```

Listado 24-9: *Respondiendo a los eventos* `addstream` *y* `icecandidate`

Tan pronto como se crea el objeto **`RTCPeerConnection`**, el navegador comienza la negociación con los servidores ICE para obtener la información requerida para acceder al ordenador local. Cuando esta información se obtiene finalmente, se informa de la aplicación a través del evento **`icecandidate`**. En la función **`prepararice()`** del Listado 24-9, la información facilitada por el evento se usa para generar un mensaje de señalización de tipo "candidate" y enviarlo al ordenador remoto. Esta vez hemos tenido que declarar la propiedad **`type`** de forma explícita porque el objeto **`candidate`** no la incluye (este es un procedimiento que tenemos que seguir para ser coherentes con el resto del proceso de señalización que hemos creado en la función **`recibido()`**).

Después de que los ordenadores concuerdan en la ruta a seguir para establecer la conexión, se comparten las transmisiones de medios. Se informa al otro ordenador de la incorporación de una nueva transmisión de medios mediante un ordenador a través del evento **`addstream`**. Cuando este evento se desencadena, la transmisión remota se debe asignar a un elemento **`<video>`**, del mismo modo que lo hemos hecho anteriormente para la transmisión del medio local procedente de la cámara y el micrófono. Con este propósito, la función **`prepararremoto()`** del Listado 24-9 crea la URL correspondiente para apuntar a la transmisión remota y la asigna como la fuente del elemento **`<video>`** con el nombre **`remotomedio`**. Esto se realiza en ambos lados de la comunicación, por lo que cada usuario puede ver su propia imagen en el lado izquierdo de la pantalla y la imagen de la otra persona a la derecha.

Lo último que necesitamos hacer para terminar la aplicación es ejecutar la función **`iniciar()`** tan pronto como se ha cargado el documento.

```
window.addEventListener("load", iniciar);
```

Listado 24-10: *Iniciando la aplicación*

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 24-2. Cree un archivo llamado webrtc.js y copie todos los códigos JavaScript desde el Listado 24-3 al Listado 24-10 dentro de este archivo. Suba ambos archivos al servidor, ejecute el servidor WebSocket como hemos explicado en el Capítulo 23 y abra el documento en su navegador. Para conectarse con otra persona, tendrá que abrir el documento en diferentes ordenadores, pero pueden estar conectados a la misma red.

IMPORTANTE: recuerde reemplazar el texto `SU_DIRECCION_IP` con la IP de su servidor. Para probar este ejemplo puede utilizar su dominio, aunque en una aplicación profesional es mejor utilizar la IP para evitar perder tiempo con el proceso de traducción DNS.

24.4 Canales de datos

La característica más revolucionaria de esta API no es la transmisión de medios sino la transmisión de datos, lo cual se realiza a través de canales de datos. Estos canales se crean sobre una conexión existente y permiten a los usuarios compartir cualquier clase de datos, que se pueden convertir a archivos o a contenido al otro lado de la conexión. La API incluye el objeto `RTCDataChannel` para representar un canal de datos y el siguiente constructor para crearlo.

createDataChannel(etiqueta, configuración**)**—Este método devuelve un objeto `RTCDataChannel`. El atributo **etiqueta** identifica el canal, y el atributo **configuración** provee un objeto con parámetros de configuración. Al ordenador remoto se le informa de la creación del canal mediante el evento `datachannel` mencionado anteriormente.

El objeto `RTCDataChannel` incluye las siguientes propiedades, métodos y eventos para configurar y trabajar con canales de datos.

label—Esta propiedad devuelve la etiqueta asignada al canal cuando se ha creado.

reliable—Esta propiedad devuelve `true` si el canal se ha declarado como fiable cuando se ha creado o `false` en caso contrario.

readyState—Esta propiedad devuelve el estado del canal.

bufferedAmount—Esta propiedad informa sobre los datos solicitados pero que aún no se han enviado al ordenador remoto. El valor que devuelve se puede usar para regular la cantidad de datos y la frecuencia de las solicitudes para no saturar la conexión.

binaryType—Esta propiedad declara el formato de los datos. Acepta dos valores: `blob` y `arraybuffer`.

send(datos**)**—Este método envía el valor del atributo **datos** al ordenador remoto. Los datos se deben especificar como una cadena de caracteres, un blob, o un ArrayBuffer.

close()—Este método cierra el canal de datos.

message—Este evento se desencadena cuando se recibe un nuevo mensaje a través de un canal de datos (nuevos datos fueron enviados por el ordenador remoto).

open—Este evento se desencadena cuando se abre el canal.

close—Este evento se desencadena cuando se cierra el canal.

error—Este evento se desencadena cuando se produce un error.

Para demostrar cómo funcionan los canales de datos, vamos a agregar una sala de chat debajo de los vídeos del ejemplo anterior, de modo que los usuarios puedan enviarse mensajes mientras conversan. El nuevo documento incluye dos pequeñas cajas en la parte superior para los vídeos, un campo de entrada, un botón para escribir y enviar los mensajes, y una caja debajo para mostrar la conversación.

```
<!DOCTYPE html>
<html lang="es">
```

```
<head>
  <meta charset="utf-8">
  <title>API WebRTC</title>
  <style>
    #local, #remoto {
      float: left;
      margin: 5px;
      padding: 5px;
      border: 1px solid #000000;
    }
    #botones {
      clear: both;
      width: 528px;
      text-align: center;
    }
    #cajadatos {
      width: 526px;
      height: 300px;
      margin: 5px;
      padding: 5px;
      border: 1px solid #000000;
    }
  </style>
  <script src="webrtc.js"></script>
</head>
<body>
  <section id="local">
    <video id="localmedio" width="250" height="200"></video>
  </section>
  <section id="remoto">
    <video id="remotomedio" width="250" height="200"></video>
  </section>
  <nav id="botones">
    <input type="button" id="botonllamar" value="Llamar">
    <input type="text" id="mensaje" size="50" required>
    <input type="button" id="botonenviar" value="Enviar">
  </nav>
  <section id="cajadatos"></section>
</body>
</html>
```

Listado 24-11: *Creando un documento para probar los canales de datos*

El código JavaScript es similar al del ejemplo anterior, pero agrega todas las funciones necesarias para crear el canal de datos y transferir mensajes desde un ordenador al otro.

```
var usuario, socket, canal, canalabierto;
function iniciar() {
  var botonllamar = document.getElementById("botonllamar");
  botonllamar.addEventListener("click", hacerllamada);
  var botonenviar = document.getElementById("botonenviar");
  botonenviar.addEventListener("click", enviardatos);
  socket = new WebSocket("ws://SU_DIRECCION_IP:12345/ws/demo.php");
  socket.addEventListener("message", recibido);
  var promesa = navigator.mediaDevices.getUserMedia({video: true});
```

```
  promesa.then(prepararcamara);
  promesa.catch(mostrarerror);
}
function mostrarerror(evento) {
  console.error(evento);
}
function prepararcamara(transmision) {
  var servidores = {"iceServers": [{"urls":
"stun:stun.l.google.com:19302"}]};
  usuario = new RTCPeerConnection(servidores);
  usuario.addEventListener("addstream", prepararremoto);
  usuario.addEventListener("icecandidate", prepararice);
  usuario.ondatachannel = function(evento) {
    canal = evento.channel;
    canal.onmessage = recibirdatos;
    canalabierto = true;
  };
  usuario.addStream(transmision);
  var video = document.getElementById("localmedio");
  video.srcObject = transmision;
  video.play();
}
function recibido(evento) {
  var mns = JSON.parse(evento.data);
  switch (mns.type) {
    case "offer":
      usuario.setRemoteDescription(new RTCSessionDescription(mns),
function() {
          usuario.createAnswer(preparardescripcion, mostrarerror);
        });
      break;
    case "answer":
      usuario.setRemoteDescription(new RTCSessionDescription(mns));
      break;
    case "candidate":
      var candidato = new RTCIceCandidate(mns.candidate);
      usuario.addIceCandidate(candidato);
  }
}
function enviarmensaje(mensaje) {
  var mns = JSON.stringify(mensaje);
  socket.send(mns);
}
function hacerllamada() {
  canal = usuario.createDataChannel("datos");
  canal.onopen = function() {
    canalabierto = true;
  };
  canal.onmessage = recibirdatos;
  usuario.createOffer(preparardescripcion, mostrarerror);
}
function preparardescripcion(descripcion) {
  usuario.setLocalDescription(descripcion, function() {
    enviarmensaje(descripcion);
  });
}
```

```
function prepararremoto(evento) {
  var video = document.getElementById("remotomedio");
  video.srcObject = evento.stream;
  video.play();
}
function prepararice(evento) {
  if (evento.candidate) {
    var mensaje = {
      type: "candidate",
      candidate: evento.candidate,
    };
    enviarmensaje(mensaje);
  }
}
function enviardatos() {
  var cajadatos = document.getElementById("cajadatos");
  if (!canalabierto) {
    cajadatos.innerHTML = "DataChannel no está disponible, intente más
adelante";
  } else {
    var mensaje = document.getElementById("mensaje").value;
    var chat = cajadatos.innerHTML;
    cajadatos.innerHTML = "Local dice: " + mensaje + "<br>";
    cajadatos.innerHTML += chat;
    canal.send(mensaje);
  }
}
function recibirdatos(evento) {
  var cajadatos = document.getElementById("cajadatos");
  var chat = cajadatos.innerHTML;
  cajadatos.innerHTML = "Remoto dice: " + evento.data + "<br>";
  cajadatos.innerHTML += chat;
}
window.addEventListener("load", iniciar);
```

Listado 24-12: *Creando una conexión que contiene un canal de datos*

El canal de datos lo tiene que crear uno de los ordenadores con el método **createDataChannel()**. El procedimiento solo se puede realizar después de que se configure la conexión y antes de que se envíe la oferta. Una manera de hacerlo es creando el canal de datos cuando se inicia una llamada. Por esta razón, decidimos declarar el canal de datos para este ejemplo en la función **hacerllamada()**. Cuando el usuario hace clic en el botón Llamar, se crea el canal de datos y se asigna a la variable **canal**. Este proceso es asíncrono, el canal se agrega a la conexión y el navegador informa a la aplicación cuándo está lista el canal a través del evento **open**. Para notificar al resto de la aplicación cuándo se puede usar el canal, cambiamos el valor de la variable **canalabierto** a **true** cuando se desencadena este evento. También respondemos al evento **message** para recibir los mensajes que proceden del otro ordenador a través de este canal.

Cuando se crea un canal en uno de los ordenadores, el otro ordenador detecta esta acción por medio del evento **datachannel**. Agregamos un listener para este evento en la función **iniciar()**. El evento devuelve un objeto con la propiedad **cannel,** la cual contiene una referencia al nuevo canal. Esta referencia se almacena en la variable **canal** y se agrega un

listener para el evento **`message`** a este lado de la conexión para recibir los mensajes que llegan a través de este canal.

Ahora, ambos ordenadores tienen un canal de datos abierto, funciones que responden al evento **`message`** y están listos para recibir y enviar mensajes. Se han creado dos funciones para este propósito: **`enviardatos()`** y **`recibirdatos()`**. Estas funciones toman el mensaje generado por el ordenador local o remoto y lo insertan en el elemento **`cajadatos`** de cada aplicación. La función **`enviardatos()`** se ejecuta cada vez que se pulsa el botón Enviar. Esta función lee la variable **`canalabierto`** para comprobar si el canal ya se ha abierto o no. Si está abierto, toma el valor de campo de entrada **`mensaje`**, lo muestra en la pantalla y lo envía al otro ordenador con el método **`send()`**. Cuando el otro ordenador recibe el mensaje, se desencadena el evento **`message`** y se llama a la función **`recibirdatos()`**. Esta función toma el valor de la propiedad **`data`** del objeto que devuelve el evento y lo muestra en la pantalla.

Hágalo usted mismo: actualice su archivo HTML con el documento del Listado 24-11 y copie el código del Listado 24-12 en el archivo webrtc.js. Suba ambos archivos a su servidor, ejecute el servidor WebSocket como hemos explicado anteriormente y abra el documento en su navegador. Inserte un mensaje en el campo de entrada y pulse el botón Enviar. Recuerde reemplazar el texto `SU_DIRECCION_IP` por la IP de su servidor.

Capítulo 25

API Web Audio

25.1 Estructura de audio

La API Web Audio ofrece las herramientas para desarrollar aplicaciones de producción de audio y procesadores de audio para juegos en la Web. Con este fin, la API utiliza una organización modular donde cada componente es un nodo (llamados *Audio Nodes*). Los nodos representan cada una de las partes del sistema de audio, desde la fuente de sonido hasta los altavoces, y se conectan entre sí para formar la estructura final que reproducirá el sonido.

Figura 25-1: *Organización básica de nodos*

La Figura 25-1 representa la estructura más básica posible: un nodo que contiene la fuente del audio y otro para el destino (altavoces, auriculares, o lo que sea que esté configurado en el ordenador para reproducir el sonido). Esta es la estructura más básica que existe, con un nodo que produce el sonido y otro que lo hace audible, pero podemos incluir más nodos con otras fuentes de audio, filtros, controles de volumen, efectos, etc. La estructura puede ser tan compleja como lo requiera nuestra aplicación.

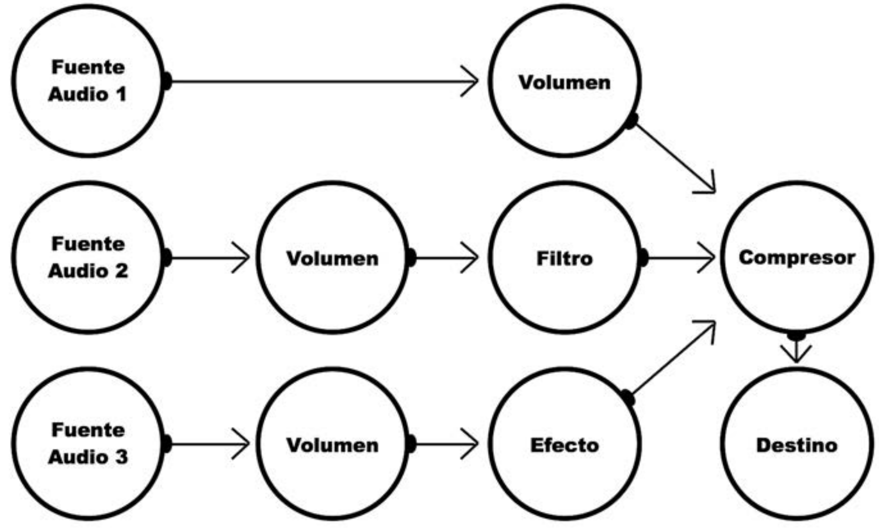

Figura 25-2: *Sistema de audio complejo*

La primera fuente de audio de la Figura 25-2 se conecta al nodo de volumen (llamado *GainNode*), el cual a su vez se conecta al nodo compresor (llamado *DynamicsCompressorNode*), que finalmente se conecta al nodo destino (altavoces, auriculares, etc.). El volumen de esta fuente de audio se modifica mediante el nodo volumen, luego se comprime, se mezcla automáticamente con el resto de los sonidos y finalmente se envía al nodo de destino para ser reproducido por la salida de audio del dispositivo. Las fuentes para los audios 2 y 3 de esta figura siguen un camino similar, pero pasan a través de un nodo filtro (llamado *BiquadFilter*) y por el nodo efecto (llamado *ConvolverNode*) antes de alcanzar el nodo compresor y de destino.

La mayoría de los nodos tienen una conexión de entrada y una de salida para recibir el sonido del nodo anterior, procesarlo y enviar el resultado al siguiente. Nuestro trabajo es crear los nodos, facilitar la información que requieren para hacer su trabajo y conectarlos.

Contexto de audio

La estructura de nodos se declara en un contexto de audio. Este contexto es similar al contexto creado para el elemento `<canvas>`. En este caso, el contexto se representa mediante un objeto de tipo `AudioContext`. La API Web Audio provee el siguiente constructor para obtener este objeto.

AudioContext()—Este constructor devuelve un objeto `AudioContext`. El objeto incluye métodos para crear cada tipo de nodo disponible, así como algunas propiedades para configurar el contexto y establecer el nodo de destino con el que acceder a la salida del dispositivo.

Las siguientes son las propiedades que se incluyen en el objeto `AudioContext` para configurar el contexto y el sistema de audio.

destination—Esta propiedad devuelve un AudioDestinationNode que representa el nodo destino. Este es el último nodo de un sistema de audio y su función es la de facilitar acceso a la salida de audio del dispositivo.

currentTime—Esta propiedad devuelve el tiempo en segundos desde que se ha creado el contexto..

activeSourceCount—Esta propiedad devuelve el número de AudioBufferSourceNode que se están reproduciendo.

sampleRate—Esta propiedad devuelve la relación en la que se está reproduciendo el audio.

listener—Esta propiedad devuelve un objeto `AudioListener` con propiedades y métodos para calcular la posición y orientación del oyente en una escena 3D.

Fuentes de audio

Los nodos más importantes son los nodos de fuentes de audio, que se describen como puntos de inicio de las estructuras presentadas en las figuras previas. Sin estas fuentes primarias de audio, no tendríamos ningún sonido que enviar a los altavoces. Estos tipos de nodos se pueden crear desde diferentes fuentes: buffers de audio en memoria, transmisiones de

medios o elementos de medios. La API facilita los siguientes métodos para obtener los objetos necesarios para representar cada una de estas fuentes.

createBufferSource()—Este método devuelve un objeto `AudioBufferSourceNode`. El objeto se crea desde un buffer de audio en memoria y facilita sus propias propiedades y métodos para reproducir y configurar el audio.

createMediaStreamSource(transmisión**)**—Este método devuelve un objeto `MediaStreamAudioSourceNode`. El objeto se crea desde una transmisión de medios. El atributo **transmisión** es una transmisión generada por métodos como `getUserMedia()` (ver Capítulo 9).

createMediaElementSource(elemento**)**—Este método devuelve un objeto `MediaElementAudioSourceNode`. El objeto se crea desde un elemento de medios. El atributo **elemento** es una referencia a un elemento `<audio>` o `<video>`.

Los métodos `createMediaStreamSource()` y `createMediaElementSource()` trabajan con transmisiones de medios y elementos de medios que tienen sus propios controles para reproducir, pausar o detener el medio, pero el método `createBufferSource()` devuelve un objeto `AudioBufferSourceNode`, el cual incluye sus propias propiedades y métodos para reproducir y configurar la fuente.

loop—Esta propiedad declara o devuelve un valor booleano que determina si el sonido asignado como la fuente del nodo se va a reproducir constantemente.

buffer—Esta propiedad declara un buffer de audio en memoria como la fuente del nodo.

start(tiempo, desplazamiento, duración**)**—Este método comienza a reproducir el sonido asignado como la fuente del nodo. El atributo **tiempo** determina cuándo se llevará a cabo la acción (en segundos). Los atributos **desplazamiento** y **duración** son opcionales. Estos atributos determinan qué parte del buffer se debe reproducir, comenzando desde el valor del **desplazamiento** y siguiendo por el tiempo determinado por la **duración** (en segundos).

stop(tiempo**)**—Este método detiene la reproducción del sonido asignado como la fuente del nodo. El atributo **tiempo** determina cuándo se llevará a cabo la acción (en segundos).

El objeto que devuelve el método `createBufferSource()` no trabaja directamente con archivos de audio. Los archivos se tienen que descargar desde el servidor, almacenar en memoria como buffers de audio y asignar al nodo de audio por medio de la propiedad `buffer`. La API provee el siguiente método para convertir los sonidos dentro de un archivo de audio en un buffer en memoria.

decodeAudioData(arraybuffer, éxito, error**)**—Este método crea un buffer de audio de forma asíncrona a partir de datos binarios. El atributo **arraybuffer** son los datos binarios que devuelve el archivo de audio (en formato ArrayBuffer). El atributo **éxito** es la función que recibirá y procesará el buffer y el atributo **error** es la función que procesará los errores.

La información de los buffers de audio se puede obtener mediante las siguientes propiedades.

duration—Esta propiedad devuelve la duración del buffer en segundos.

length—Esta propiedad devuelve la extensión del buffer en cuadros (llamados sample frames).

numberOfChannels—Esta propiedad devuelve el número de canales disponibles en el buffer.

sampleRate—Esta propiedad devuelve la relación del buffer en cuadros por segundo.

Conectando nodos

Antes de crear nuestra primera estructura de audio, tenemos que estudiar cómo conectar los nodos. Los nodos de audio tienen una conexión de salida y entrada que nos permiten establecer la ruta que tiene que seguir el sonido. La API incluye dos métodos para construir y gestionar esta red de nodos.

connect(nodo**)**—Este método conecta la salida de un nodo con la entrada de otro. La sintaxis es **`salida.connect(entrada)`**, donde **`salida`** es una referencia al nodo que facilita la salida y **`entrada`** es una referencia al nodo que recibe el audio desde la salida.

disconnect()—Este método desconecta la salida del nodo. La sintaxis es **`salida.disconnect()`**, donde **`salida`** es una referencia al nodo que se desconectará.

25.2 Aplicaciones de audio

Normalmente los sonidos son parte de códigos JavaScript complejos, como los necesarios para crear videojuegos en 3D o aplicaciones de producción de audio, pero por fines didácticos vamos a usar un documento sencillo para simplificar la creación e implementación de nodos de audio.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Web Audio</title>
  <script src="audio.js"></script>
</head>
<body>
  <section>
    <button type="button" id="boton"
disabled="true">Reproducir</button>
  </section>
</body>
</html>
```

Listado 25-1: *Creando un documento para reproducir sonidos*

Como hemos explicado antes, una estructura de nodos básica requiere un nodo para la fuente y otro para el destino. El nodo destino lo devuelve la propiedad del contexto **`destination`**, pero la fuente de audio requiere un poco de trabajo. Tenemos que descargar

el archivo de audio, convertir su contenido a un buffer de audio, usar el buffer como fuente del nodo y finalmente conectar ambos nodos para escuchar el sonido a través de los altavoces.

Para nuestro ejemplo, vamos a descargar un archivo WAV usando Ajax y el objeto **`XMLHttpRequest`** (ver Capítulo 21).

```
var contexto;
function iniciar() {
  var mibuffer;
  var boton = document.getElementById("boton");
  boton.addEventListener("click", function() {
    reproducir(mibuffer);
  });
  contexto = new AudioContext();
  var url = "disparo.wav";
  var solicitud = new XMLHttpRequest();
  solicitud.responseType = "arraybuffer";
  solicitud.addEventListener("load", function() {
    if (solicitud.status == 200) {
      contexto.decodeAudioData(solicitud.response, function(buffer) {
        mibuffer = buffer;
        boton.disabled = false;
      });
    }
  });
  solicitud.open("GET", url, true);
  solicitud.send();
}
function reproducir(mibuffer) {
  var nodofuente = contexto.createBufferSource();
  nodofuente.buffer = mibuffer;
  nodofuente.connect(contexto.destination);
  nodofuente.start(0);
}
window.addEventListener("load", iniciar);
```

Listado 25-2: *Reproduciendo un buffer de audio*

Como siempre, tenemos nuestra función **`iniciar()`** con la que iniciar la aplicación. La función comienza obteniendo una referencia al botón del documento y agregando un listener para el evento **`click`** con el fin de ejecutar la función **`reproducir()`** cada vez que se pulsa el botón. Luego, creamos el contexto de audio con el constructor **`AudioContext()`** y lo almacenamos en la variable **`contexto`**. El archivo de audio disparo.wav que queremos reproducir se descarga a continuación usando Ajax.

El proceso de descarga para este archivo es el mismo que aplicamos en los ejemplos del Capítulo 21, excepto que esta vez el tipo de respuesta se declara como **`arraybuffer`** porque necesitamos este tipo de datos para crear el buffer de audio en memoria. Una vez que se ha descargado el archivo y la propiedad **`status`** de la solicitud devuelve el valor 200 (OK), se crea el buffer de forma asíncrona a partir del valor de la propiedad **`response`** usando el método **`decodeAudioData()`**. Después de que los datos se convierten en un buffer de audio, el método llama a una función para informar del resultado. En esta función asignamos el buffer producido por el método a la variable **`mibuffer`** y habilitamos el botón Reproducir para permitir al usuario reproducir el sonido.

Todo el sistema de audio, incluidos los nodos y las conexiones, se tiene que reconstruir cada vez que queremos reproducir el sonido. La función `iniciar()` se encarga de descargar el archivo y convertir su contenido en un buffer de audio en memoria, pero el resto se realiza mediante la función `reproducir()`. Cada vez que se pulsa el botón Reproducir, se crea el sistema de audio con esta función. Primero, se crea el nodo de la fuente de audio con el método `createBufferSource()` y luego el buffer de audio se asigna como el valor de la propiedad `buffer` del nodo. Este proceso conecta el nodo con el buffer de audio en memoria. Después de que se ha configurado el nodo, se conecta al nodo destino por medio del método `connect()` y el sonido que representa el nodo se reproduce finalmente con el método `start()`.

El botón Reproducir se inhabilita al comienzo, pero se habilita con la función `iniciar()` después de que los datos necesarios para crear el sistema de audio están listos. Este es el proceso que hemos decidido seguir para simplificar este ejemplo. Una vez que se habilita el botón, cada vez que se pulsa, el sistema de audio se crea de nuevo y se reproduce el sonido del archivo disparo.wav.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 25-1 y un archivo JavaScript llamado audio.js con el código del Listado 25-2. Puede descargar el archivo disparo.wav desde nuestro sitio web o utilizar su propio archivo de audio. Suba todos los archivos a su servidor y abra el documento en su navegador. Haga clic en el botón Reproducir para reproducir el sonido.

Bucles y tiempos

El método `start()` al final de la función `reproducir()` en el último ejemplo se ha llamado con el valor `0`. Cuando este valor se declara como `0` o es menor que el tiempo actual del contexto, el sonido comienza a reproducirse inmediatamente. Una vez que se reproduce, el método `start()` no se puede llamar nuevamente para reproducir la misma fuente de audio; tenemos que recrear el sistema de audio completo. Esta es la razón por la que construimos el sistema de audio en una función aparte. A pesar de estas restricciones, hay varias maneras de reproducir un sonido varias veces sin tener que volver a llamar a la función `reproducir()`. Una alternativa es reproducir la fuente de audio en un bucle declarando la propiedad `loop` con el valor `true`.

```
function reproducir(mibuffer) {
  var nodofuente = contexto.createBufferSource();
  nodofuente.buffer = mibuffer;
  nodofuente.loop = true;
  nodofuente.connect(contexto.destination);
  nodofuente.start(0);
  nodofuente.stop(contexto.currentTime + 3);
}
```

Listado 25-3: *Reproduciendo la fuente de audio en un bucle*

El sonido se reproduce de forma indefinida hasta que se ejecuta el método `stop()`. En el código del Listado 25-3, usamos este método para detener el bucle después de 3 segundos. El tiempo se declara usando el valor que devuelve la propiedad `currentTime`. Si agregamos el

valor 3 al tiempo actual, ordenamos al navegador detener la reproducción del sonido 3 segundos después de que se ejecute la instrucción.

Hágalo usted mismo: reemplace la función `reproducir()` del Listado 25-2 con la nueva función del Listado 25-3. Suba el archivo audio.js a su servidor y abra el documento del Listado 25-1 en el navegador.

Nodos de audio

Cada uno de los nodos de audio se llama *Audio Node*. Los únicos nodos que necesitamos para producir un sonido son el nodo de la fuente de audio y el nodo destino, creados en los ejemplos anteriores, pero estos no son los únicos nodos disponibles. La API ofrece métodos para construir un variedad de nodos con los que procesar, analizar e incluso crear sonidos.

createAnalyser()—Este método crea un AnalyserNode. Estos tipos de nodos facilitan acceso a información de la fuente de audio. El objeto devuelto incluye varias propiedades y métodos con este propósito. Se usa a menudo para visualizar transmisiones de audio.

createGain()—Este método crea un GainNode. Estos tipos de nodos declaran el volumen de la señal de audio. El objeto devuelto ofrece la propiedad `gain` para declarar el volumen. Acepta valores entre `0.0` y `1.0`.

createDelay(máximo**)**—Este método crea un DelayNode. Este tipo de nodos declara un retraso para la fuente de audio. El objeto devuelto facilita la propiedad `delayTime` para declarar el retraso en segundos. El atributo **máximo** es opcional y declara el tiempo máximo para el retraso en segundos.

createBiquadFilter()—Este método crea un BiquadFilterNode. Estos tipos de nodos aplican filtros a la señal de audio. El objeto devuelto ofrece propiedades, métodos y también varias constantes para determinar las características del filtro.

createWaveShaper()—Este método crea un WaveShaperNode. Esto tipos de nodos aplican un efecto de distorsión basado en una curva de modelado. El objeto devuelto ofrece la propiedad `curve` para declarar el tipo de curva (tipo Float32Array).

createPanner()—Este método crea un PannerNode. Estos tipos de nodos se usan para determinar la posición, orientación y velocidad del sonido en un espacio tridimensional. El efecto producido depende de los valores actuales declarados para el oyente. El objeto devuelto facilita varios métodos y propiedades para configurar los parámetros del nodo.

createConvolver()—Este método crea un ConvolverNode. Estos tipos de nodos aplican un efecto de circunvolución a la señal de audio basado en una respuesta de impulso. El objeto devuelto ofrece dos propiedades: `buffer` y `normalize`. La propiedad `buffer` es necesaria para declarar el buffer que vamos a usar como la respuesta de impulso y la propiedad `normalize` recibe un valor booleano para declarar si se escalará la respuesta de impulso.

createChannelSplitter(salidas**)**—Este método crea un ChannelSplitterNode. Estos tipos de nodos generan diferentes salidas para cada canal de la señal de audio. El atributo **salidas** declara el número de salidas a generar.

createChannelMerger(entradas**)**—Este método crea un ChannelMergerNode. Estos tipos de nodos combinan canales desde múltiples señales de audio en una sola. El atributo **entradas** declara el número de entradas a mezclar.

createDynamicsCompressor()—Este método crea un DynamicsCompressorNode. Estos tipos de nodos aplican un efecto de compresión a la señal de audio. El objeto devuelto facilita varias propiedades para configurar el efecto.

createOscillator()—Este método crea un OscillatorNode. Estos tipos de nodos generan formas de onda para síntesis de audio. El objeto devuelto facilita varias propiedades y métodos para configurar la onda.

IMPORTANTE: la aplicación de algunos de estos métodos requiere conocimientos en ingeniería de audio. El tema va más allá del propósito de este libro, pero a continuación estudiaremos algunas aplicaciones de los métodos necesarios para la construcción de aplicaciones de audio y también de videojuegos en 2D y 3D.

AudioParam

Además de las propiedades y métodos que todo nodo ofrece para configurar el sonido, la API incluye un objeto adicional para declarar parámetros específicos. El objeto `AudioParam` es como una unidad de control conectada al nodo para ajustar sus valores.

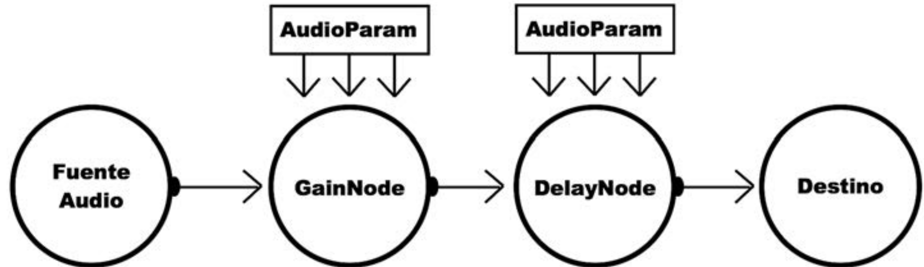

Figura 25-3: *Nodos controlados por las propiedades y métodos de AudioParam*

Este objeto puede asignar valores de forma inmediata o se puede configurar para hacerlo más adelante. Las siguientes son las propiedades y los métodos que se facilitan para este propósito.

value—Esta propiedad declara el valor del parámetro tan pronto como se define.

setValueAtTime(valor, inicio**)**—Este método declara el valor del parámetro en un momento determinado. Los atributos **valor** e **inicio** declaran el nuevo valor del parámetro y el tiempo en segundos, respectivamente.

linearRampToValueAtTime(valor, fin**)**—Este método cambia el valor del parámetro gradualmente desde el valor actual al especificado en los atributos. Los atributos **valor** y **fin** declaran el nuevo valor y el tiempo de finalización del proceso en segundos, respectivamente.

exponentialRampToValueAtTime(valor, fin**)**—Este método cambia el valor actual del parámetro exponencialmente al valor especificado por los atributos. Los atributos **valor** y **fin** declaran el nuevo valor y el tiempo de finalización del proceso en segundos, respectivamente.

setTargetAtTime(objetivo, inicio, constante**)**—Este método cambia el valor del parámetro exponencialmente al valor del atributo **objetivo**. El atributo **inicio** declara el tiempo de inicio del proceso y el atributo **constante** determina el ritmo al cual el valor anterior se incrementará hasta alcanzar el nuevo.

setValueCurveAtTime(valores, inicio, duración**)**—Este método cambia el valor del parámetro por valores arbitrarios seleccionados desde un array declarado por el atributo **valores** (tipo Float32Array). Los atributos **inicio** y **duración** declaran el tiempo de inicio del proceso y la duración en segundos, respectivamente.

cancelScheduledValues(inicio**)**—Este método cancela todos los cambios previamente programados en el momento especificado por el atributo **inicio**.

GainNode

Lo primero que necesitamos hacer cuando reproducimos un sonido es subir o bajar el volumen. Para este propósito, vamos a introducir un GainNode entre el nodo de la fuente de audio y el nodo de destino creados en ejemplos anteriores.

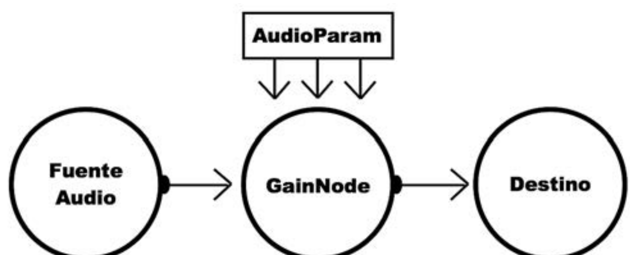

Figura 25-4: *Controlando el volumen con GainNode*

Cuando agregamos un nuevo nodo a un sistema de audio, la estructura básica permanece igual; solo tenemos que crear el nuevo nodo, definir los valores de configuración y conectarlo al resto de la estructura. En este caso, el nodo de la fuente de audio se tiene que conectar al GainNode y el GainNode al nodo destino.

```
function reproducir(mibuffer) {
  var nodofuente = contexto.createBufferSource();
  nodofuente.buffer = mibuffer;

  var nodovolumen = contexto.createGain();
  nodovolumen.gain.value = 0.2;
  nodofuente.connect(nodovolumen);
  nodovolumen.connect(contexto.destination);
  nodofuente.start(0);
}
```

Listado 25-4: *Bajando el volumen con un GainNode*

La función **`reproducir()`** del Listado 25-4 introduce un GainNode en nuestro sistema de audio. El nodo se crea mediante el método **`createGain()`** y luego, usando la propiedad **`value`** del objeto **`AudioParam`**, se asigna un valor de **`0.2`** a la propiedad **`gain`** del nodo. Los posibles valores para esta propiedad van desde **`0.0`** a **`1`**. Por defecto, el valor de **`gain`** es **`1`**, por lo que el volumen en este ejemplo se reduce un 20 %.

Las conexiones hechas al final del código incluyen el nuevo GainNode. Primero, el nodo de la fuente de audio se conecta al GainNode (**`nodofuente.connect(nodovolumen)`**) y luego el GainNode se conecta al nodo destino (**`nodovolumen. connect(context.destination)`**).

El valor de la propiedad **`gain`** para este ejemplo se ha declarado con un número fijo con la propiedad **`value`** porque el archivo de audio contiene solo el sonido breve de un disparo, pero podríamos haber usado cualquiera de los métodos de **`AudioParam`** en su lugar para incrementar o reducir el volumen en momentos diferentes (como al final de una canción, por ejemplo). Se pueden usar métodos como **`exponentialRampToValueAtTime()`** en combinación con la propiedad **`duration`** de los buffers para mezclar canciones, con lo que se reduce el volumen de una pista y se incrementa el de la siguiente.

Hágalo usted mismo: reemplace la función **`reproducir()`** del Listado 25-2 con la nueva función del Listado 25-4. Suba el archivo audio.js, el documento HTML del Listado 25-1 y el archivo de audio a su servidor, abra el documento en su navegador y pulse el botón Reproducir.

DelayNode

El propósito de un DelayNode es retrasar el sonido durante el tiempo que se especifica con la propiedad **`delayTime`** (y el objeto **`AudioParam`**). En el siguiente ejemplo, vamos a introducir un nodo para reproducir el sonido del disparo un segundo después.

```
function reproducir(mibuffer) {
  var nodofuente = contexto.createBufferSource();
  nodofuente.buffer = mibuffer;

  nodoretardo = contexto.createDelay();
  nodoretardo.delayTime.value = 1;
  nodofuente.connect(nodoretardo);
  nodoretardo.connect(contexto.destination);
  nodofuente.start(0);
}
```

Listado 25-5: *Introduciendo un retraso*

La función **`reproducir()`** del Listado 25-5 reemplaza el GainNode del ejemplo anterior con un DelayNode. El retraso se declara en `1` segundo mediante la propiedad **`value`** del objeto **`AudioParam`** y las conexiones se declaran siguiendo la misma ruta anterior: el nodo de la fuente de audio se conecta al DelayNode y este nodo se conecta al nodo destino.

Un DelayNode produce mejores resultados cuando se combina con otros nodos, como en el siguiente ejemplo.

```
function reproducir(mibuffer) {
  var nodofuente = contexto.createBufferSource();
  nodofuente.buffer = mibuffer;
  nodoretardo = contexto.createDelay();
  nodoretardo.delayTime.value = 0.3;

  nodovolumen = contexto.createGain();
  nodovolumen.gain.value = 0.2;

  nodofuente.connect(nodoretardo);
  nodoretardo.connect(nodovolumen);
  nodovolumen.connect(contexto.destination);

  nodofuente.connect(contexto.destination);
  nodofuente.start(0);
}
```

Listado 25-6: *Obteniendo un efecto de eco con el DelayNode*

En la función `reproducir()` del Listado 25-6, se agrega un GainNode en el medio de la estructura para reducir el volumen del sonido retrasado y generar un efecto de eco. Se declaran dos rutas para la fuente de audio. En una de las rutas, el nodo de la fuente de audio se conecta al DelayNode, el DelayNode al GainNode y el GainNode al nodo destino. El sonido en esta ruta tendrá un volumen bajo y se reproducirá con un retraso de `0.3` segundos. La segunda ruta para la fuente es una conexión directa al nodo destino. El sonido en esta ruta se reproducirá de inmediato a todo volumen.

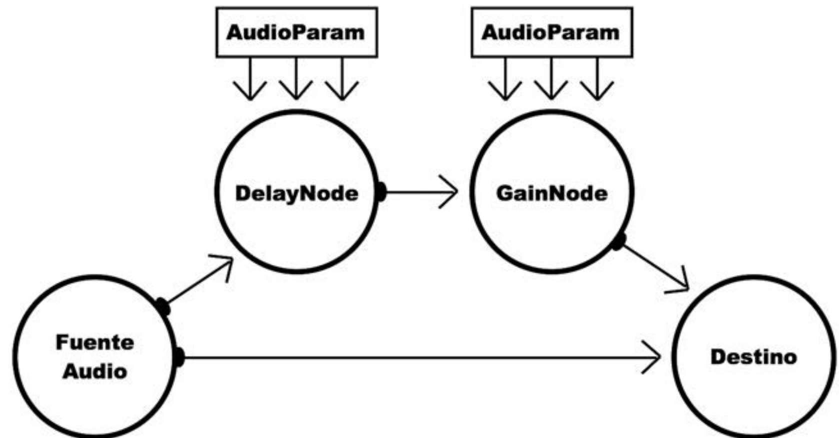

Figura 25-5: *Dos rutas para la misma fuente de audio*

Hágalo usted mismo: reemplace la función `reproducir()` del Listado 25-2 con las nuevas funciones de los Listados 25-5 o 25-6 para probar cada ejemplo. Suba los archivos a su servidor, abra el documento en su navegador y pulse el botón Reproducir. Debería escuchar dos sonidos, uno después del otro, simulando un efecto de eco.

BiquadFilterNode

El BiquadFilterNode nos permite aplicar filtros a la señal de audio. El nodo se crea mediante el método **createBiquadFilter()**, y el tipo de filtro se selecciona mediante la propiedad **type**. Los valores disponibles para esta propiedad son "lowpass", "highpass", "bandpass", "lowshelf", "highshelf", "peaking", "notch", y "allpass". El nodo también incluye las propiedades **frequency**, **Q** y **gain** para configurar el filtro. El efecto producido por los valores de estas propiedades en el filtro depende del tipo de filtro aplicado.

```
function reproducir(mibuffer) {
  var nodofuente = contexto.createBufferSource();
  nodofuente.buffer = mibuffer;

  var nodofiltro = contexto.createBiquadFilter();
  nodofiltro.type = "highpass";
  nodofiltro.frequency.value = 1000;

  nodofuente.connect(nodofiltro);
  nodofiltro.connect(contexto.destination);
  nodofuente.start(0);
}
```

Listado 25-7: *Agregando un filtro con el BiquadFilterNode*

En el ejemplo del Listado 25-7, declaramos el tipo del filtro como "highpass" y especificamos la frecuencia de corte con el valor **1000**. Esto cortará algunas frecuencias del espectro haciendo que el disparo suene como si se emitiera con un arma de juguete.

El valor de la propiedad **frequency** se declara mediante la propiedad **value** del objeto **AudioParam**. Este procedimiento se puede reemplazar por cualquier método de **AudioParam** para variar la frecuencia a través del tiempo.

Hágalo usted mismo: reemplace la función **reproducir()** del Listado 25-2 con la nueva función del Listado 25-7. Suba los archivos a su servidor, abra el documento en su navegador y pulse Reproducir.

IMPORTANTE: el efecto producido por los valores de las propiedades **frequency**, **Q** y **gain** depende del filtro seleccionado. Algunos valores no producirán ningún efecto. Para más información, lea la especificación de la API. Los enlaces están disponibles en nuestro sitio web.

DynamicsCompressorNode

Los compresores suavizan el audio, reduciendo el volumen de sonidos altos y elevando el de sonidos bajos. Estos tipos de nodos normalmente se conectan al final de un sistema de audio con el propósito de coordinar los niveles de las fuentes para crear un sonido unificado. La API Web Audio incluye el DynamicsCompressorNode para producir este efecto en la señal de audio. El nodo tiene varias propiedades para limitar y controlar la compresión: **threshold** (decibelios), **knee** (decibelios), **ratio** (valores desde 1 a 20), **reduction** (decibelios), **attack** (segundos) y **release** (segundos).

```
function reproducir(mibuffer) {
  var nodofuente = contexto.createBufferSource();
  nodofuente.buffer = mibuffer;

  var nodocompresor = contexto.createDynamicsCompressor();
  nodocompresor.threshold.value = -60;
  nodocompresor.ratio.value = 10;

  nodofuente.connect(nodocompresor);
  nodocompresor.connect(contexto.destination);
  nodofuente.start(0);
}
```

Listado 25-8: *Agregando un compresor dinámico*

El código del Listado 25-8 muestra cómo crear y configurar el DynamicsCompressorNode, pero este nodo no siempre se usa de esta manera. En general, estos tipos de nodos se implementan al final de sistemas de audio elaborados que incluyen varias fuentes de audio.

Hágalo usted mismo: reemplace la función **`reproducir()`** del Listado 25-2 con la nueva función del Listado 25-8. Suba los archivos a su servidor, abra el documento en su navegador y pulse Reproducir.

ConvolverNode

El ConvolverNode aplica un efecto de circunvolución a una señal de audio. Se usa frecuentemente para simular diferentes espacios acústicos y lograr distorsiones de sonido complejas, como una voz en el teléfono, por ejemplo. El efecto se logra calculando una respuesta de impulso con un segundo archivo de audio. Usualmente, este archivo de audio es una grabación efectuada en un espacio acústico real, similar al que queremos simular. Debido a estos requerimientos, para implementar el efecto tenemos que descargar al menos dos archivos de audio, uno para la fuente y otro para la respuesta de impulso, como en el siguiente ejemplo.

```
var contexto;
var misbuffers = [];
function iniciar() {
  var boton = document.getElementById("boton");
  boton.addEventListener("click", function() {
    reproducir();
  });
  contexto = new AudioContext();
  cargarbuffers("disparo.wav", 0);
  cargarbuffers("garaje.wav", 1);

  var control = function() {
    if (misbuffers.length >= 2) {
      boton.disabled = false;
    } else {
      setTimeout(control, 200);
    }
  };
```

```
  control();
}
function cargarbuffers(url, id) {
  var solicitud = new XMLHttpRequest();
  solicitud.responseType = "arraybuffer";
  solicitud.addEventListener("load", function(){
    if (solicitud.status == 200) {
      contexto.decodeAudioData(solicitud.response, function(buffer) {
        misbuffers[id] = buffer;
      });
    }
  });
  solicitud.open("GET", url, true);
  solicitud.send();
}
function reproducir() {
  var nodofuente = contexto.createBufferSource();
  nodofuente.buffer = misbuffers[0];

  var nodoconvolver = contexto.createConvolver();
  nodoconvolver.buffer = misbuffers[1];

  nodofuente.connect(nodoconvolver);
  nodoconvolver.connect(contexto.destination);
  nodofuente.start(0);
}
window.addEventListener("load", iniciar);
```

Listado 25-9: *Aplicando un efecto de circunvolución*

En el código del Listado 25-9, hemos recreado todo el código JavaScript, incluido un pequeño cargador para poder descargar los dos archivos de audio necesarios para el efecto (disparo.wav y garaje.wav). Esta vez hemos tenido que mover el código Ajax a una nueva función para descargar estos archivos uno por uno y crear la función **control()** para controlar el proceso de descarga.

Al comienzo de la función **iniciar()**, la función **cargarbuffers()** se llama dos veces para descargar los archivos. Se crea un buffer a partir de cada archivo y se almacena en el array **misbuffers**. Este proceso se controla mediante la función **control()** y cuando el tamaño del array es igual o mayor que **2**, se activa el botón Reproducir.

La construcción del sistema de audio en la función **reproducir()** es similar a ejemplos anteriores, excepto que esta vez dos nodos requieren buffers de audio, por lo que asignamos el ítem correspondiente del array **misbuffers** a la propiedad **buffer** de cada nodo y finalmente se conectan ambos nodos.

Hágalo usted mismo: copie el código del Listado 25-9 dentro del archivo audio.js, suba todos los archivos a su navegador, abra el documento del Listado 25-1 en su navegador y pulse Reproducir.

PannerNode y sonido 3D

Las librerías gráficas en tres dimensiones son herramientas que han logrado un nivel de realismo increíble hoy en día. La recreación de objetos reales en la pantalla de un ordenador

ha alcanzado altos niveles de perfección y realismo, y gracias a WebGL, esta increíble experiencia ahora está disponible para la Web. Sin embargo, los gráficos no son suficientes para recrear el mundo real; también necesitamos sonidos que cambian junto con los cambios en la posición de la fuente y varían según la velocidad de la fuente y su orientación. PannerNode es un nodo diseñado específicamente para simular estos efectos. Este nodo es similar a otros nodos de audio, pero los sonidos se deben configurar considerando los parámetros de un mundo en 3D. El objeto que representa el nodo incluye las siguientes propiedades y métodos para este propósito.

panningModel—Esta propiedad especifica el algoritmo que vamos a utilizar para posicionar el sonido en la escena 3D. Los valores disponibles son "equalpower" y "HRTF".

distanceModel—Esta propiedad especifica el algoritmo a usar para reducir el volumen del sonido de acuerdo al movimiento de la fuente. Los valores disponibles son "linear", "inverse" y "exponential".

refDistance—Esta propiedad especifica un valor de referencia para calcular la distancia entre la fuente y el oyente. Puede ser útil para adaptar el nodo a la escala de nuestra escena 3D. El valor por defecto es `1`.

maxDistance—Esta propiedad especifica la distancia máxima entre la fuente del sonido y el oyente. Después de este límite, el sonido mantendrá sus valores actuales.

rolloffFactor—Esta propiedad especifica el ritmo al que se reducirá el volumen.

coneInnerAngle—Esta propiedad especifica el ángulo para fuentes de audio direccionales. Dentro de este ángulo, el volumen no se reduce

coneOuterAngle—Esta propiedad especifica el ángulo para fuentes de audio direccionales. Fuera de este ángulo, el volumen se reduce a un valor determinado por la propiedad **`coneOuterGain`**.

setPosition(x, y, z**)**—Este método declara la posición de la fuente de audio relativa al oyente. Los atributos **x**, **y** y **z** declaran el valor de cada coordenada.

setOrientation(x, y, z**)**—Este método declara la dirección de una fuente de audio direccional. Los atributos **x**, **y** y **z** declaran un vector de dirección.

setVelocity(x, y, z**)**—Este método declara la velocidad y dirección de la fuente de audio. Los atributos **x**, **y** y **z** declaran un vector de dirección que representa la dirección y también la velocidad de la fuente.

Los sonidos en una escena 3D están relacionados con el oyente. Debido a que solo existe un oyente, sus características se definen mediante el contexto de audio y no mediante un nodo particular. Se puede acceder al objeto que representa al oyente a través de la propiedad **`listener`** del contexto y contar con las siguientes propiedades y métodos para la configuración.

dopplerFactor—Esta propiedad especifica un valor para configurar el cambio de tono para el efecto Doppler.

speedOfSound—Esta propiedad especifica la velocidad del sonido en metros por segundo. Se usa para calcular el cambio Doppler. El valor por defecto es **`343.3`**.

setPosition(x, y, z**)**—Este método establece la posición del oyente. Los atributos **x**, **y** y **z** declaran el valor de cada coordenada.

setOrientation(x, y, z, xSuperior, ySuperior, zSuperior**)**—Este método establece la dirección en la que está orientado el oyente. Los atributos **x**, **y** y **z** declaran un vector de dirección para la parte frontal del oyente, mientras que los atributos **xSuperior**, **ySuperior** y **zSuperior** declaran un vector de dirección para la parte superior del oyente.

setVelocity(x, y, z**)**—Este método establece la velocidad y dirección del oyente. Los atributos **x**, **y** y **z** declaran un vector de dirección que representa la dirección y la velocidad del oyente.

Veremos todo esto en una aplicación 3D usando la librería Three.js estudiada en el Capítulo 12. El documento que hemos usado hasta el momento en este capítulo se tiene que modificar para que incluya el archivo three.min.js y el elemento **<canvas>**.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Web Audio</title>
  <script src="three.min.js"></script>
  <script src="audio.js"></script>
</head>
<body>
  <section>
    <canvas id="canvas" width="500" height="400"></canvas>
  </section>
</body>
</html>
```

Listado 25-10: *Creando un documento para probar sonidos en 3D*

El código para este ejemplo dibuja un cubo en la pantalla que podemos mover hacia adelante y hacia atrás con la rueda del ratón.

```
var contexto, nodopanner, renderer, escena, camara, malla;
function iniciar() {
  var canvas = document.getElementById("canvas");
  var ancho = canvas.width;
  var altura = canvas.height;
  renderer = new THREE.WebGLRenderer({canvas: canvas});
  renderer.setClearColor(0xFFFFFF);
  escena = new THREE.Scene();
  camara = new THREE.PerspectiveCamera(45, ancho/altura, 0.1, 10000);
  camara.position.set(0, 0, 150);

  var geometria = new THREE.BoxGeometry(50, 50, 50);
  var material = new THREE.MeshPhongMaterial({color: 0xCCCCFF});
  malla = new THREE.Mesh(geometria, material);
  malla.rotation.y = 0.5;
  malla.rotation.x = 0.5;
```

```
  escena.add(malla);
  var luz = new THREE.SpotLight(0xFFFFFF, 1);
  luz.position.set(0, 100, 250);
  escena.add(luz);

  contexto = new AudioContext();
  contexto.listener.setPosition(0, 0, 150);

  var url = "motor.wav";
  var solicitud = new XMLHttpRequest();
  solicitud.responseType = "arraybuffer";
  solicitud.addEventListener("load", function() {
    if (solicitud.status == 200) {
      contexto.decodeAudioData(solicitud.response, function(buffer) {
        reproducir(buffer);
        canvas.addEventListener("mousewheel", mover, false);
        renderer.render(escena, camara);
      });
    }
  });
  solicitud.open("GET", url, true);
  solicitud.send();
}
function reproducir(mibuffer) {
  var nodofuente = contexto.createBufferSource();
  nodofuente.buffer = mibuffer;
  nodofuente.loop = true;
  nodopanner = contexto.createPanner();
  nodopanner.refDistance = 100;
  nodofuente.connect(nodopanner);
  nodopanner.connect(contexto.destination);
  nodofuente.start(0);
}
function mover(evento) {
  malla.position.z += evento.wheelDeltaY / 5;
  nodopanner.setPosition(malla.position.x, malla.position.y,
malla.position.z);
  renderer.render(escena, camara);
}
window.addEventListener("load", iniciar);
```

Listado 25-11: *Calculando la posición del sonido en una escena 3D*

La función **iniciar()** del Listado 25-11 inicializa todos los elementos de la escena 3D y descarga el archivo motor.wav con el sonido para nuestro cubo. Después de crear el contexto de audio con el constructor **audioContext()**, la posición del oyente se declara con el método **setPosition()**. Esto se realiza en el proceso de inicialización porque en este ejemplo el oyente mantiene la misma posición todo el tiempo, pero seguramente se deberá modificar en otro tipo de aplicaciones.

Después de que se descargue el archivo y se cree el buffer de audio, agregamos un listener para el evento **mousewheel** para ejecutar la función **mover()** cada vez que se gira la rueda del ratón. Esta función actualiza la posición del cubo en el eje **z** de acuerdo con la dirección en que se ha movido la rueda y luego declara las nuevas coordenadas para la fuente del audio. El efecto hace que el cubo parezca la fuente real del sonido.

En la función `reproducir()` se crea el PannerNode y se incluye en el sistema de audio. En nuestro ejemplo, usamos la propiedad `refDistance` de este objeto para declarar los valores del nodo relativos a la escala de nuestra escena 3D.

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 25-10, copie el código del Listado 25-11 dentro del archivo audio.js y descargue el archivo motor.wav desde nuestro sitio web. Suba todos los archivos a su servidor, abra el documento en su navegador y gire la rueda del ratón para mover el cubo. Debería escuchar el sonido cambiar a medida que el cubo se aleja o se acerca.

AnalyserNode

Sería muy difícil implementar todas las herramientas provistas por la API Web Audio en un ambiente profesional si no pudiéramos visualizar los resultados en la pantalla. Para lograr esto, la API incluye el AnalyserNode. Este nodo implementa un algoritmo llamado *FFT* (fast Fourier transform) para convertir la forma de onda de la señal de audio en un array de valores que representan magnitud versus frecuencia en un período de tiempo determinado. Los valores devueltos se pueden usar para analizar la señal o crear gráficos con los que mostrar los valores en la pantalla. El AnalyserNode facilita las siguientes propiedades y métodos para obtener y procesar la información.

fftSize—Esta propiedad especifica el tamaño de la FFT (el tamaño del bloque de datos a analizar). El valor debe ser una potencia de 2 (por ejemplo, 128, 256, 512, 1024, etc.).

frequencyBinCount—Esta propiedad devuelve la cantidad de valores de frecuencia provistos por la FFT.

minDecibels—Esta propiedad especifica el valor mínimo de decibelios para la FFT.

maxDecibels—Esta propiedad especifica el valor máximo de decibelios para la FFT.

smoothingTimeConstant—Esta propiedad declara un período de tiempo en el que el analizador obtendrá un valor promedio de las frecuencias. Acepta un valor entre `0` y `1`.

getFloatFrequencyData(array**)**—Este método obtiene los datos de la frecuencia actual de la señal de audio y los almacena en el atributo **array** como valores decimales. El atributo es una referencia a un array de valores decimales que ya se ha creado.

getByteFrequencyData(array**)**—Este método obtiene los datos de la frecuencia actual de la señal de audio y los almacena en el atributo **array** como bytes sin signo. El atributo es una referencia a un array de bytes sin signo que ya se ha creado.

getByteTimeDomainData(array**)**—Este método obtiene los datos de la forma de onda actual y los almacena en el atributo **array** como valores decimales. El atributo es una referencia a un array de valores decimales que ya se ha creado.

Los datos producidos por estas propiedades y métodos se pueden usar para actualizar un gráfico en pantalla y mostrar la evolución de la señal de audio. Para demostrar cómo implementar un AnalyserNode vamos a usar un elemento `<canvas>` y dibujar el sonido producido por un elemento `<video>`.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Web Audio</title>
  <style>
    section {
      float: left;
    }
  </style>
  <script src="audio.js"></script>
</head>
<body>
  <section>
    <video id="medio" width="483" height="272" controls>
      <source src="trailer.mp4">
      <source src="trailer.ogg">
    </video>
  </section>
  <section>
    <canvas id="canvas" width="500" height="272"></canvas>
  </section>
</body>
</html>
```

Listado 25-12: *Creando un documento para experimentar con el AnalyserNode*

Como hemos explicado antes, cuando la fuente es un elemento de medios, se tiene que crear el nodo de la fuente de audio con el método **`createMediaElementSource()`**. En el siguiente código, aplicamos este y el método **`createAnalyser()`** para obtener los nodos que necesitamos para nuestro sistema de audio.

```
var canvas, contexto, nodoanalizador;
function iniciar() {
  var video = document.getElementById("medio");
  var elemento = document.getElementById("canvas");
  canvas = elemento.getContext("2d");
  contexto = new AudioContext();

  var nodofuente = contexto.createMediaElementSource(video);
  nodoanalizador = contexto.createAnalyser();
  nodoanalizador.fftSize = 512;
  nodoanalizador.smoothingTimeConstant = 0.9;

  nodofuente.connect(nodoanalizador);
  nodoanalizador.connect(contexto.destination);
  mostrargraficos();
}
function mostrargraficos() {
  var datos = new Uint8Array(nodoanalizador.frequencyBinCount);
  nodoanalizador.getByteFrequencyData(datos);

  canvas.clearRect(0, 0, 500, 400);
  canvas.beginPath();
```

```
  for (var f = 0; f < nodoanalizador.frequencyBinCount; f++) {
    canvas.fillRect(f * 5, 272 - datos[f], 3, datos[f]);
  }
  canvas.stroke();
  requestAnimationFrame(mostrargraficos);
}
window.addEventListener("load", iniciar);
```

Listado 25-13: *Dibujando un gráfico de sonido en un elemento* `<canvas>`

En este ejemplo, el tamaño de la FFT se declara como **`512`** y se establece un período de tiempo de **`0.9`** con la propiedad **`smoothingTimeConstant`** para limitar el tamaño de los datos obtenidos y suavizar el gráfico en la pantalla. Después de los nodos se crean, configuran y conectan, el sistema de audio está listo para ofrecer la información de la señal de audio. La función **`mostrargraficos()`** está a cargo de procesar estos datos. Esta función crea un array vacío de tipo Uint8Array y el tamaño determinado por la propiedad **`frequencyBinCount`**, y luego llama al método **`getByteFrequencyData()`** para asignar al array valores de la señal de audio. Debido al tipo de array, los valores almacenados serán enteros de **`0`** a **`256`** (enteros de 8 bits). En el bucle **`for`**, a continuación, usamos estos valores para calcular el tamaño de las barras que representan las frecuencias correspondientes y dibujarlas en el lienzo.

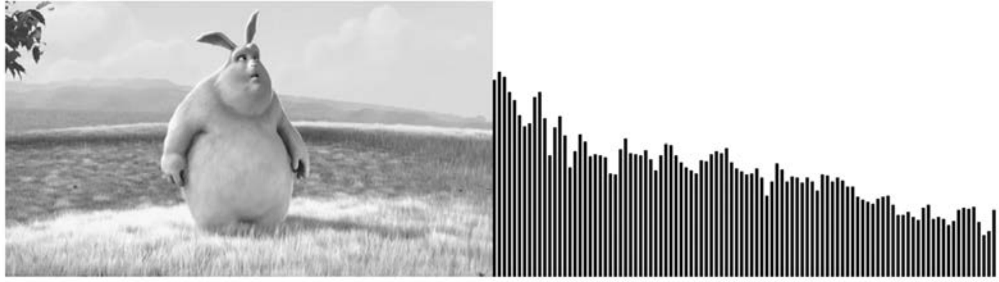

Figura 25-6: *Gráfico de barras para el vídeo creado con el AnalyserNode*
© Derechos Reservados 2008, Blender Foundation / www.bigbuckbunny.org

Hágalo usted mismo: cree un nuevo archivo HTML con el documento del Listado 25-12, copie el código del Listado 25-13 dentro del archivo audio.js, abra el documento en su navegador y reproduzca el vídeo. Puede descargar los vídeos trailer.mp4 y trailer.ogg desde nuestro sitio web.

Lo básico: las variables en JavaScript se definen normalmente sin tener en cuenta el tipo de datos con los que van a trabajar, pero algunos métodos aceptan solo valores en formatos específicos. El constructor **`Uint8Array()`**, implementado en el ejemplo del Listado 25-13, es solo uno de los constructores que introduce el lenguaje para permitir crear nuevos tipos de array con los que satisfacer los requerimientos de algunas API. Para obtener más información sobre el tema, visite nuestro sitio web y siga los enlaces de este capítulo.

Capítulo 26

API Web Workers

26.1 Procesamiento paralelo

JavaScript se ha convertido en la herramienta principal para la creación de aplicaciones en la Web, pero desde su creación, el lenguaje estaba destinado a procesar un solo código a la vez. La incapacidad de procesar múltiples códigos simultáneamente reduce la eficacia y limita el alcance de esta tecnología, lo que vuelve imposible la simulación de aplicaciones de escritorio en la Web. Web Workers es una API diseñada con el propósito de convertir a JavaScript en un lenguaje de procesamiento paralelo y resolver este problema.

Workers

Un worker (trabajador) es un código JavaScript que ejecuta procesos en segundo plano mientras el resto de la aplicación se continúa ejecutando y es capaz de responder al usuario. El mecanismo para crear el worker y conectarlo a la aplicación principal es sencillo; el worker se construye en un archivo JavaScript aparte y los códigos se comunican entre sí a través de mensajes. El siguiente es el constructor que facilita la API para crear un objeto `Worker`.

Worker(URL**)**—Este constructor devuelve un objeto `Worker`. El atributo **URL** es la URL del archivo con el código (worker) que se ejecutará en segundo plano.

Enviando y recibiendo mensajes

El mensaje enviado al worker desde el código principal es la información que queremos que se procese, y los mensajes enviados de regreso por el worker representan los resultados de ese procesamiento. Para enviar y recibir estos mensajes, la API aprovecha la técnicas implementadas en la API Web Messaging (ver Capítulo 22). Los siguientes son los métodos y eventos que se necesitan para realizar este proceso.

postMessage(mensaje**)**—Este método envía un mensaje hacia o desde el worker. El atributo **mensaje** puede ser cualquier valor JavaScript, como una cadena de caracteres o un número, y también datos binarios, como un objeto `File` o un ArrayBuffer que representa el mensaje a transmitir.

message—Este evento se desencadena cuando se recibe un mensaje desde el otro código. Al igual que el método `postMessage()`, se puede aplicar en el worker o en el código principal. El evento genera un objeto con la propiedad `data` con el contenido del mensaje.

El siguiente es un documento sencillo que vamos a utilizar para enviar nuestro nombre al worker e imprimir la respuesta. Incluso un ejemplo básico como este requiere al menos tres archivos: el documento principal, el código principal y el archivo con el código para el worker.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>API Web Workers</title>
  <link rel="stylesheet" href="webworkers.css">
  <script src="webworkers.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <label for="nombre">Nombre: </label><br>
      <input type="text" name="nombre" id="nombre"><br>
      <button type="button" id="boton">Enviar</button>
    </form>
  </section>
  <section id="cajadatos"></section>
</body>
</html>
```

Listado 26-1: *Creando un documento para experimentar con Web Workers*

El documento del Listado 26-1 requiere las siguientes reglas para diseñar el formulario y la caja en la que mostraremos los mensajes.

```
#cajaformulario {
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos {
  float: left;
  width: 500px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
```

Listado 26-2: *Definiendo los estilos para las cajas (webworkers.css)*

El código JavaScript para el documento tiene que enviar al worker la información que queremos que se procese. Este código también tiene que ser capaz de recibir la respuesta.

```
var worker, cajadatos;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  var boton = document.getElementById("boton");
```

```
  boton.addEventListener("click", enviar);
  worker = new Worker("worker.js");
  worker.addEventListener("message", recibido);
}
function enviar() {
  var nombre = document.getElementById("nombre").value;
  worker.postMessage(nombre);
}
function recibido(evento) {
  cajadatos.innerHTML = evento.data;
}
window.addEventListener("load", iniciar);
```

Listado 26-3: *Cargando el worker (webworkers.js)*

El Listado 26-3 presenta el código para nuestro documento (el que va dentro del archivo webworkers.js). Después de la creación de las referencias necesarias para el elemento **`cajadatos`** y el botón, se crea el objeto **`Worker`**. El constructor **`Worker()`** toma el archivo worker.js con el código del worker y devuelve un objeto **`Worker`** con esta referencia. Cada interacción con este objeto será una interacción con el código de este archivo.

Después de obtener este objeto, agregamos un listener para el evento **`message`** para poder recibir mensajes que proceden del worker. Cuando se recibe un mensaje, se llama a la función **`recibido()`** y el valor de la propiedad **`data`** (el mensaje) se muestra en pantalla.

El otro lado de la comunicación lo controla la función **`enviar()`**. Cuando el usuario pulsa el botón Enviar, se obtiene el valor del campo de entrada **`nombre`** y se envía al worker usando el método **`postMessage()`**.

Con las funciones **`recibido()`** y **`enviar()`** a cargo de las comunicaciones, el código ya puede enviar mensajes al worker y procesar sus respuestas. Ahora tenemos que preparar el worker.

```
addEventListener("message", recibido);

function recibido(evento) {
  var respuesta = "Su nombre es " + evento.data;
  postMessage(respuesta);
}
```

Listado 26-4: *Creando el worker (worker.js)*

Del mismo modo que el código del Listado 26-3 es capaz de recibir mensajes que proceden del worker, el código del worker tiene que ser capaz de recibir mensajes que proceden del código principal. Por esta razón, en la primera línea del Listado 26-4 agregamos un listener al worker para el evento **`message`**. Cada vez que este evento se desencadena (se recibe un mensaje), se ejecuta la función **`recibido()`**. En esta función, el valor de la propiedad **`data`** se agrega a un texto predefinido y el resultado se envía de regreso al código principal y usa nuevamente el método **`postMessage()`**.

Hágalo usted mismo: compare los códigos de los Listados 26-3 y 26-4 (el código principal y el worker). Estudie cómo se lleva a cabo el proceso de comunicación y cómo se aplican los mismo métodos y eventos en ambos

códigos para este propósito. Cree los archivos usando los Listados 26-1, 26-2, 26-3 y 26-4, súbalos a su servidor, abra el documento en su navegador, escriba su nombre en el campo de entrada y pulse el botón. Debería ver el mensaje que envía de regreso el worker en la pantalla.

Este worker es, por supuesto, elemental. Realmente no se procesa nada, solo se crea una cadena de caracteres a partir del mensaje recibido que se envía de forma inmediata de regreso como respuesta. Sin embargo, este ejemplo es útil para entender cómo se comunican entre sí los códigos y cómo aprovechar esta API.

IMPORTANTE: a pesar de sus simplicidad, hay varios puntos importantes que debe considerar a la hora de crear sus propios workers. Los mensajes son la única forma de comunicarse con los workers. Además, los workers no pueden acceder al documento o manipular elementos HTML, y no se puede acceder a las funciones y variables del código principal desde los workers. Los workers son como código enlatado, que solo tiene permitido procesar la información recibida a través de mensajes y enviar el resultado con el mismo mecanismo.

Errores

Los workers son poderosas unidades de procesamiento. Podemos usar funciones, métodos nativos de JavaScript y API completas desde dentro de un worker. Considerando lo complejo que se puede volver un worker, la API incorpora un evento para controlar los errores producidos por un worker y devolver toda la información disponible al respecto.

error—Este evento se desencadena mediante el objeto `Worker` en el código principal cada vez que ocurre un error en el worker. El evento genera un objeto con tres propiedades para informar acerca del error: `message`, `filename` y `lineno`. La propiedad `message` devuelve el mensaje de error. Es una cadena de caracteres que nos indica lo que ha pasado. La propiedad `filename` devuelve el nombre del archivo que contiene el código que ha causado el error. Esto es útil cuando el worker carga los archivos externos, tal como veremos más adelante. Y la propiedad `lineno` devuelve el número de línea donde ha ocurrido el error.

El siguiente código muestra los errores que devuelve un worker.

```
var worker, cajadatos;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  var boton = document.getElementById("boton");
  boton.addEventListener("click", enviar);
  worker = new Worker("worker.js");
  worker.addEventListener("error", mostrarerror);
}
function enviar() {
  var nombre = document.getElementById("nombre").value;
  worker.postMessage(nombre);
}
```

```
function mostrarerror(evento) {
  cajadatos.innerHTML = "ERROR: " + evento.message + "<br>";
  cajadatos.innerHTML += "Archivo: " + evento.filename + "<br>";
  cajadatos.innerHTML += "Línea: " + evento.lineno;
}
window.addEventListener("load", iniciar);
```

Listado 26-5: *Respondiendo al evento* `error` *(webworkers.js)*

El código JavaScript del Listado 26-5 es similar al código principal del Listado 26-3. Este código construye un worker, pero solo usa el evento **`error`** porque esta vez no necesitamos recibir respuestas del worker, solo controlar los errores producidos por el mismo. Por supuesto, el código no realiza ninguna función importante, pero demuestra cómo se devuelven los errores y la clase de información facilitada en estas circunstancias. Para probarlo, podemos generar un error desde el worker llamando a una función que no existe.

```
addEventListener("message", recibido);

function recibido(evento){
  prueba();
}
```

Listado 26-6: *Produciendo un error (worker.js)*

Cuando el código del Listado 26-6 recibe un mensaje, se ejecuta la función **`recibido()`** y se llama a la función **`prueba()`**, lo que genera un error. Tan pronto como ocurre el error, el evento **`error`** se desencadena en el código principal y se llama a la función **`mostrarerror()`**, lo que muestra en la pantalla los valores de las tres propiedades que facilita el evento.

Hágalo usted mismo: para este ejemplo, usamos el documento HTML y las reglas CSS de los Listados 26-1 y 26-2. Copie el código del Listado 26-5 en el archivo webworkers.js y el código del Listado 26-6 dentro del archivo worker.js. Abra el documento del Listado 26-1 en su navegador y envíe cualquier valor al worker desde el formulario para activar el proceso. El error que devuelve el worker se mostrará en la pantalla.

Finalizando workers

Los workers son unidades de código especiales que funcionan constantemente en segundo plano y esperan información que se ha de procesar. Normalmente, estos servicios no se requieren todo el tiempo y, por lo tanto, es una buena práctica detenerlos o finalizar su ejecución si ya no los necesitamos. La API facilita dos métodos diferentes con este propósito.

terminate()—Este método finaliza el worker desde el código principal.

close()—Este método finaliza el worker desde dentro del mismo worker.

Cuando se finaliza un worker, todos los procesos que se están ejecutando en ese momento se anulan y se desecha cualquier tarea pendiente en el bucle de eventos. Para probar ambos métodos, vamos a crear una pequeña aplicación que trabaja exactamente igual que nuestro primer ejemplo, pero que también responde a dos comandos: "cerrar1" y "cerrar2". Si los textos "cerrar1" o "cerrar2" se envían desde el formulario, el worker finaliza mediante el código principal o el código del worker usando los métodos **`terminate()`** o **`close()`**, respectivamente.

```
var worker, cajadatos;
function iniciar() {
  cajadatos = document.getElementById("cajadatos");
  var boton = document.getElementById("boton");
  boton.addEventListener("click", enviar);
  worker = new Worker("worker.js");
  worker.addEventListener("message", recibido);
}
function enviar() {
  var nombre = document.getElementById("nombre").value;
  if (nombre == "cerrar1") {
    worker.terminate();
    cajadatos.innerHTML = "Worker Terminado";
  } else {
    worker.postMessage(nombre);
  }
}
function recibido(evento) {
  cajadatos.innerHTML = evento.data;
}
window.addEventListener("load", iniciar);
```

Listado 26-7: *Terminando el worker desde el código principal (webworkers.js)*

La única diferencia entre el código del Listado 26-7 y el del Listado 26-3 es la adición de una instrucción **`if`** para comprobar la inserción del comando "cerrar1". Si este comando se inserta en el formulario, se ejecuta el método **`terminate()`** y se muestra un mensaje en pantalla que indica que el worker ha finalizado. Por otro lado, si el texto es diferente del comando esperado, se envía como un mensaje al worker.

El código del worker realiza una tarea similar. Si el mensaje recibido es igual a "cerrar2", el worker se finaliza a sí mismo con el método **`close()`**. En caso contrario, envía un mensaje de regreso.

```
addEventListener("message", recibido);

function recibido(evento) {
  if (evento.data == "cerrar2") {
    postMessage("Worker Terminado");
    close();
  } else {
    var respuesta = "Su nombre es " + evento.data;
    postMessage(respuesta);
  }
}
```

Listado 26-8: *Finalizando el worker desde su interior*

Hágalo usted mismo: utilice el mismo documento HTML y las reglas CSS de los Listados 26-1 y 26-2. Copie el código del Listado 26-7 dentro del archivo webworkers.js y el código del Listado 26-8 dentro del archivo worker.js. Abra el documento en su navegador y, mediante el formulario, envíe los comandos "cerrar1" o "cerrar2". Después de esto, el worker ya no enviará ninguna respuesta.

API síncronas

Los workers pueden presentar limitaciones a la hora de trabajar con el documento principal y acceder a sus elementos, pero cuando se trata de procesamiento y funcionalidad, como mencionamos anteriormente, están listos para la tarea. Por ejemplo, dentro de un worker podemos usar métodos convencionales como **`setTimeout()`** o **`setInterval()`**, cargar información adicional desde servidores con Ajax y también acceder a otras API para crear poderosas aplicaciones. Esta última posibilidad es la más prometedora de todas, pero presenta una trampa: tenemos que incluir una implementación diferente de las API disponibles para workers.

Cuando estudiamos algunas API en capítulos anteriores, la implementación que indicamos fue la llamada *asíncrona*. La mayoría de las API tienen versiones asíncronas y síncronas disponibles. Estas versiones diferentes de la misma API realizan las mismas tareas pero usan métodos específicos de acuerdo a la forma en la que se procesan. Las API asíncronas son útiles cuando las operaciones realizadas requieren mucho tiempo para ser completadas y consumen recursos que el documento principal necesita en ese momento. Las operaciones asíncronas se llevan a cabo en segundo plano mientras el código principal se sigue ejecutando sin interrupción. Debido a que los workers funcionan al mismo tiempo que el código principal, ya son asíncronos, y estos tipos de operaciones ya no son necesarias. En consecuencia, los workers tienen que implementar las versiones síncronas de estas API.

IMPORTANTE: varias API ofrecen versiones síncronas, como la API File y la API IndexedDB, pero actualmente algunas de ellas se encuentran en desarrollo o no son estables. Visite los enlaces en nuestro sitio web para obtener más información al respecto.

Importando código JavaScript

Algo que vale la pena mencionar es la posibilidad de cargar archivos JavaScript externos desde un worker. Un worker puede contener todo el código necesario para realizar cualquier tarea que necesitemos, pero debido a que se pueden crear varios workers para el mismo documento, existe la posibilidad de que algunas partes de sus códigos se vuelvan redundantes. Para solucionar este problema, podemos seleccionar estas partes, ponerlas en un único archivo y luego cargar ese archivo desde cada worker que lo requiera con el siguiente método.

importScripts(archivo**)**—Este método carga un archivo JavaScript externo para incorporar código adicional al worker. El atributo **archivo** indica la ruta del archivo a incluir.

La manera en la que trabaja el método **`importScripts()`** es similar a la de los métodos que facilitan otros lenguajes, como **`include()`** de PHP, por ejemplo. El código del archivo se incorpora al worker como si fuera parte de su propio código. Para usar este método, tenemos

que declararlo al comienzo del worker. El código para el worker no estará listo hasta que estos archivos se hayan cargado completamente.

```
importScripts("mascodigos.js");

addEventListener("message", recibido);
function recibido(evento) {
  prueba();
}
```

Listado 26-9: *Cargando códigos JavaScript externos desde un worker*

El código del Listado 26-9 no es un código funcional, pero es un ejemplo de cómo debemos usar el método **importScripts()**. En esta situación hipotética, el archivo mascodigos.js que contiene la función **prueba()** se carga en cuanto termina de cargarse el archivo del worker. Después de esto, la función **prueba()** (y cualquier otra función dentro del archivo mascodigos.js) queda disponible para el resto del código del worker.

Workers compartidos

El worker que hemos estudiado hasta ahora se llama *Worker Dedicado* (Dedicated Worker). Este tipo de workers solo responde al código principal desde el cual se ha creado. Existe otro tipo de worker llamado *Worker Compartido* (Shared Worker), que responde a varios documentos en el mismo origen. Trabajar con varias conexiones significa que podemos compartir el mismo worker desde diferentes ventanas, pestañas, o marcos y mantenerlos a todos actualizados y sincronizados. La API facilita un objeto para representar Workers Compartidos llamado **SharedWorker**. El siguiente es el constructor que necesitamos para crear estos objetos.

SharedWorker(URL**)**—Este constructor reemplaza al constructor **Worker()** usado para crear Workers Dedicados. El atributo **URL** declara la ruta del archivo JavaScript con el código para el worker. Se puede agregar un segundo atributo para especificar el nombre del worker.

Las conexiones se realizan a través de puertos y estos puertos se pueden almacenar dentro del worker para usar como referencia. Para trabajar con Workers Compartidos y puertos, esta parte de la API incorpora nuevas propiedades, eventos y métodos.

port—Cuando se construye el objeto **SharedWorker**, se crea un nuevo puerto para el documento y se asigna a la propiedad **port**. Esta propiedad se usará luego para referenciar el puerto y comunicarnos con el worker.

connect—Este evento comprueba la existencia de nuevas conexiones desde dentro del worker. El evento se desencadena cada vez que un documento inicia una conexión con el worker. Es útil para llevar un control de todas las conexiones disponibles para el worker (para referenciar todos los documentos que lo están usando).

start()—Este método está disponible desde objetos **MessagePort** (uno de los objetos devueltos durante la construcción de un Worker Compartido) y su función es la de comenzar

a despachar los mensajes recibidos a través de un puerto. Después de la construcción del objeto **SharedWorker**, se debe llamar a este método para iniciar la conexión.

El constructor **SharedWorker()** devuelve un objeto **SharedWorker** y un objeto **MessagePort** con el valor del puerto a través del cual se llevará a cabo la conexión con el worker. La comunicación con el Worker Compartido se debe realizar a través del puerto referenciado por el valor de la propiedad **port**.

El documento de nuestro ejemplo incluye un iframe para cargar otro documento en la misma ventana. Los dos documentos, el documento principal y el documento dentro del iframe, compartirán el mismo worker.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Web Workers</title>
  <link rel="stylesheet" href="webworkers.css">
  <script src="webworkers.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <label for="nombre">Nombre: </label>
      <input type="text" name="nombre" id="nombre">
      <button type="button" id="boton">Enviar</button>
    </form>
  </section>
  <section id="cajadatos">
    <iframe id="iframe" src="iframe.html" width="500"
height="350"></iframe>
  </section>
</body>
</html>
```

Listado 26-10: *Creando un documento para experimentar con Workers Compartidos*

El documento del iframe debe incluir un elemento para mostrar la información y un elemento **<script>** para incluir el archivo iframe.js que contendrá el código para comunicarse con el worker.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>iframe</title>
  <script src="iframe.js"></script>
</head>
<body>
  <section id="cajadatos"></section>
</body>
</html>
```

Listado 26-11: *Creando el documento para el iframe (iframe.html)*

Cada documento tiene su propio código JavaScript para iniciar la conexión con el worker y procesar sus respuestas. Estos códigos tienen que construir el objeto **SharedWorker** y usar el puerto referenciado por el valor de la propiedad **port** para enviar y recibir mensajes. El siguiente es el código correspondiente al documento principal.

```
var worker;
function iniciar() {
  var boton = document.getElementById("boton");
  boton.addEventListener("click", enviar);
  worker = new SharedWorker("worker.js");
  worker.port.addEventListener("message", recibido);
  worker.port.start();
}
function recibido(evento) {
  alert(evento.data);
}
function enviar() {
  var nombre = document.getElementById("nombre").value;
  worker.port.postMessage(nombre);
}
window.addEventListener("load", iniciar);
```

Listado 26-12: *Conectándose con el worker desde el documento principal (webworkers.js)*

Cada documento que quiera trabajar con un Worker Compartido tiene que crear el objeto **SharedWorker** y configurar la conexión con el worker. En el código del Listado 26-12, el objeto se construye con el archivo worker.js y luego la comunicación se establece a través del puerto correspondiente con la propiedad **port**.

Después de agregar un listener para el evento **message** con el fin de recibir respuestas desde el worker, se llama al método **start()** para comenzar a despachar mensajes. La conexión con un Worker Compartido no se establece hasta que se ejectua este método.

La función **enviar()** es similar al ejemplo anterior, pero esta vez la comunicación se realiza a través del valor de la propiedad **port**.

El código para el iframe es muy similar.

```
function iniciar() {
  var worker = new SharedWorker("worker.js");
  worker.port.addEventListener("message", recibido);
  worker.port.start();
}
function recibido(evento) {
  var cajadatos = document.getElementById("cajadatos");
  cajadatos.innerHTML = evento.data;
}
window.addEventListener("load", iniciar);
```

Listado 26-13: *Conectándose al worker desde el iframe (iframe.js)*

En ambos códigos, el objeto **SharedWorker** se crea referenciando el mismo archivo (worker.js) y la conexión se establece usando la propiedad **port** (aunque a través de puertos diferentes). La única diferencia entre el código del documento principal y el código del iframe es cómo se procesa

esa respuesta. En el documento principal, la función **`recibido()`** muestra una ventana emergente con un mensaje (ver Listado 26-12), mientras que dentro del iframe, la respuesta se imprime como un simple texto dentro del elemento **`cajadatos`** (ver Listado 26-13).

Es hora de ver cómo el Worker Compartido gestiona cada conexión y envía los mensajes de regreso al documento correcto. Recuerde que solo tenemos un worker para ambos documentos (de aquí el nombre de Worker Compartido). Toda solicitud de conexión al worker se tiene que diferenciar y almacenar para poderse usar más adelante como referencia. En nuestro worker, vamos a almacenar las referencias a los puertos de cada documento en un array llamado **`puertos`**.

```
var puertos = new Array();
addEventListener("connect", conectar);

function conectar(evento) {
  puertos.push(evento.ports[0]);
  evento.ports[0].onmessage = enviar;
}
function enviar(evento) {
  for (var f = 0; f < puertos.length; f++) {
    puertos[f].postMessage("Su nombre es " + evento.data);
  }
}
```

Listado 26-14: *Respondiendo desde el Worker Compartido (worker.js)*

Este procedimiento es similar al que usamos con Workers Dedicados, pero esta vez tenemos que considerar a qué documento vamos a responder, porque puede haber varios conectados con el worker al mismo tiempo. Para este propósito, el evento **`connect`** facilita el array **`ports`** con el valor del puerto recién creado (el array solo contiene este valor ubicado en el índice 0).

Cada vez que un código solicita una conexión al worker, se desencadena el evento **`connect`**. En el código del Listado 26-14, este evento llama a la función **`conectar()`**. En esta función realizamos dos operaciones. Primero, el valor del puerto se toma de la propiedad **`ports`** (índice 0) y se almacena en el array **`puertos`** (inicializado al comienzo del worker), y segundo, la propiedad del evento **`onmessage`** se define para este puerto en particular y la función **`enviar()`** se declara para responder al mismo. En consecuencia, cada vez que un mensaje se envía al worker desde el código principal, independientemente del documento al que pertenece, se ejecuta la función **`enviar()`** en el worker. En esta función usamos un bucle **`for`** para obtener todos los puertos abiertos para este worker y enviamos un mensaje a cada documento conectado. El proceso es el mismo que usamos para Workers Dedicados, pero esta vez se responden varios documentos en lugar de uno solo.

Hágalo usted mismo: para probar este ejemplo, debe crear varios archivos y subirlos a su servidor. Cree un archivo HTML con el documento del Listado 26-10. Este documento cargará el mismo archivo webworkers.css usado en este capítulo, el archivo webworkers.js con el código del Listado 26-12 y el archivo iframe.html como la fuente del iframe con el código del Listado 26-11. También tiene que crear un archivo llamado worker.js para el worker con el código del Listado 26-14. Una vez que todos estos archivos se almacenan y

se suben al servidor, abra el documento principal en su navegador. Use el formulario para enviar un mensaje al worker y ver cómo ambos documentos (el documento principal y el documento en el iframe) procesan la respuesta.

Índice

#

áxim(), 283, 287
áximo(), 279

:

::cue, 373
:first-child, 94, 97
:first-of-type, 95
:full-screen, 389
:hover, 134
:in-range, 355
:invalid, 354
:last-child, 94, 97
:not(), 95, 97
:nth-child(), 94, 96
:only-child, 94, 97
:optional, 355
:out-of-range, 355
:required, 355
:valid, 354

@

@font-face, 102
@keyframes, 137
@media, 200, 202

<

<!DOCTYPE>, 7, 19
<a>, 40, 43
<address>, 38, 39
<article>, 26, 30
<aside>, 26, 29
<audio>, 361
<b>, 37
<base>, 21
<blockquote>, 50
<body>, 7, 20, 21

, 36
<button>, 58
<canvas>, 391
<cite>, 38
<code>, 38, 39
<colgroup>, 54
<data>, 38
<datalist>, 58, 71
<dd>, 48, 49
<details>, 50, 51
<div>, 25, 160
<dl>, 48, 49
<dt>, 48, 49
<em>, 37
<fieldset>, 58
<figcaption>, 45, 47
<figure>, 45, 47
<footer>, 26, 30, 31
<form>, 56, 345
<h1>, 34
<head>, 7, 20, 21
<header>, 26, 27
<html>, 6, 20
<i>, 37
<iframe>, 554
<img>, 45, 46, 142, 217, 418
<input>, 57, 497
<label>, 58, 61
<link>, 22, 23, 85
<main>, 25, 29
<mark>, 37, 38
<meta>, 21, 22, 23, 205
<meter>, 58, 72
<nav>, 26, 28, 40, 163
, 48, 49
<option>, 70
<output>, 58
<p>, 34, 35
<picture>, 45, 218
<pre>, 34, 35, 39
<progress>, 58, 72
<script>, 8, 22, 245
<section>, 26
<select>, 57, 70
<small>, 37, 39
<source>, 218, 360, 362
<span>, 35, 88
<strong>, 37
<style>, 22, 85
<summary>, 50, 51
<table>, 52
<tbody>, 54
<td>, 52
<textarea>, 57

<tfoot>, 54
<th>, 52, 53
<thead>, 54
<time>, 38, 39
<title>, 21, 22
<tr>, 52
<track>, 370
<u>, 37
, 48, 163
<video>, 357, 359, 425, 447
<wbr>, 36

A

abort, 484, 504, 543
abort(), 539
abs(), 300
accept-charset, 57
activeCues, 375
activeSourceCount, 586
add(), 315, 483
addColorStop(), 394
addCue(), 378
addEventListener(), 324
addIceCandidate(), 569
addstream, 571
addStream(), 570
addTextTrack(), 378
alert(), 242, 303, 337
align-content, 180, 189
align-items, 180, 184
align-self, 180, 187
altKey, 329, 332
AmbientLight(), 440
animation, 136
animation-delay, 136
animation-direction, 136
animation-duration, 136
animation-fill-mode, 136
animation-iteration-count, 136
animation-name, 136
animation-timing-function, 136
append(), 545
appendChild(), 321
arc(), 397, 399
Array(), 282
assert(), 337
assign(), 304
AudioContext(), 586
autocomplete, 76, 80
autofocus, 76
autoplay, 357, 361

B

back(), 527
backface-visibility, 133
background, 111, 113, 312
background-attachment, 111
background-clip, 111
background-color, 96, 110, 201
background-image, 110, 111
background-origin, 111
background-position, 110
background-repeat, 110, 112
background-size, 110
beginPath(), 396
bezierCurveTo(), 397, 401
binaryType, 579
blur, 537
blur(), 127
Boolean(), 281
border, 114, 312
border-color, 113
border-image, 118
border-image-outset, 118
border-image-repeat, 118
border-image-slice, 118
border-image-source, 118
border-image-width, 118
border-radius, 115
border-style, 113
border-width, 113
bound(), 494
BoxGeometry(), 433
box-shadow, 120
box-sizing, 207, 209
break, 262
brightness(), 127
buffer, 587
buffered, 363
bufferedAmount, 561, 579
button, 329

C

cancelScheduledValues(), 593
canplaythrough, 364
canPlayType(), 364
catch, 340
catch(), 381
ceil(), 300, 302
character entities, 31
checkValidity(), 345
CircleGeometry(), 434
class, 32, 92

classList, 315, 316
className, 315, 316
clear, 141, 145
clear(), 337, 475
clearData(), 508
clearInterval(), 306
clearRect(), 393
clearWatch(), 519, 524
click, 242, 323
clientHeight, 314
clientWidth, 314
clientX, 329, 330, 467
clientY, 329, 330, 467
clip(), 396, 399
close, 562, 579
close(), 561, 569, 579, 609
closePath(), 396, 398
color, 84, 104, 312
column-count, 155
column-fill, 155
column-gap, 155
column-rule, 157
column-rule-color, 157
column-rule-style, 157
column-rule-width, 157
columns, 155
column-span, 155
column-width, 155
complete, 484
concat(), 288, 293
coneInnerAngle, 599
coneOuterAngle, 599
confirm(), 303
connect, 612
connect(), 588
contains(), 316
contenteditable, 55
continue, 262
continue(), 490
contrast(), 127
controls, 357, 358, 361
coords, 520
copy, 410
cos(), 301
count, 492
createAnalyser(), 591, 603
createAnswer(), 570
createBiquadFilter(), 591, 596
createBufferSource(), 587
createChannelMerger(), 592
createChannelSplitter(), 591
createConvolver(), 591
createDataChannel(), 579
createDelay(), 591
createDynamicsCompressor(), 592
createElement(), 321
createGain(), 591, 594
createIndex(), 483
createLinearGradient(), 394
createMediaElementSource(), 587, 603
createMediaStreamSource(), 587
createObjectStore(), 483, 487
createObjectURL(), 503
createOffer(), 570
createOscillator(), 592
createPanner(), 591
createPattern(), 413
createRadialGradient(), 394
createWaveShaper(), 591
credential, 573
crossorigin, 46
crossOrigin, 416
cssFloat, 313
ctrlKey, 329, 332
cues, 375
currentTime, 363, 586
customError, 351
CylinderGeometry(), 433

D

darker, 410
data, 553
datachannel, 571
DataTransfer, 512
Date(), 295, 296
decodeAudioData(), 587
decodeURIComponent(), 269
default, 260, 371
delete(), 483, 490
deleteDatabase(), 481
deleteIndex(), 484
deleteObjectStore(), 483
destination, 586
destination-atop, 409
destination-in, 409
destination-out, 409
destination-over, 409
direction, 492
DirectionalLight(), 440
disabled, 76
disconnect(), 588
display, 139, 231
distanceModel, 599
do while, 261
document, 302, 307

dopplerFactor, 599
download, 44
drag, 507
dragend, 507
dragenter, 507
draggable, 510
dragleave, 507
dragover, 507
dragstart, 507
drawImage(), 411, 515
drop, 507
dropEffect, 508
drop-shadow(), 127
duration, 363, 588

E

effectAllowed, 508
em, 214
enabled, 383
enableHighAccuracy, 522
encodeURIComponent(), 269, 270
enctype, 56
ended, 363, 364, 385
endsWith(), 283, 287
error, 363, 364, 484, 486, 487, 504, 543, 562, 579, 608
ErrorEvent, 339
every(), 289, 291
exitFullscreen(), 387
exitPointerLock(), 463, 467
exp(), 301
exponentialRampToValueAtTime(), 593

F

fftSize, 602
File, 500, 516
FileReader(), 498
files, 507
fill(), 396, 398
fillRect(), 392
fillStyle, 394
fillText(), 403
filter, 127
filter(), 289, 290
flex, 172, 173, 205
flex-basis, 173, 176
flex-direction, 179
flex-grow, 173, 176
flex-shrink, 173, 176
flex-wrap, 180, 188
float, 141
floor(), 300, 302
focus, 537
font, 99, 313, 403
font-family, 98, 99, 102
font-size, 84, 87, 98, 99
font-style, 99
font-weight, 98
for, 260
FormData(), 545
formnovalidate, 76, 78
forward(), 527
frequencyBinCount, 602
FTP, 14
fullscreenchange, 387
fullscreenElement, 387
fullscreenEnabled, 387
fullscreenerror, 387
function, 263

G

gatheringchange, 571
GET, 73, 540
get(), 483
getAudioTracks(), 383
getByteFrequencyData(), 602
getByteTimeDomainData(), 602
getContext(), 392
getCurrentPosition(), 519
getData(), 508
getDate(), 295
getDay(), 295
getElementById(), 307, 309
getElementsByClassName(), 307
getElementsByName(), 307
getElementsByTagName(), 308, 309
getFloatFrequencyData(), 602
getFullYear(), 295, 297
getHours(), 295
getImageData(), 414
getItem(), 473
getMilliseconds(), 296
getMinutes(), 295
getMonth(), 295, 297
getSeconds(), 296
getTime(), 296, 299
getUserMedia(), 381, 428, 574
getVideoTracks(), 383
globalAlpha, 394
globalCompositeOperation, 409
go(), 527
grayscale(), 127

H

high, 72
history, 302, 527
hsl(), 104, 105
hsla(), 104
HTTP, 73
hue-rotate(), 127

I

icecandidate, 571
icechange, 571
iceServers, 573
iceState, 569
IcosahedronGeometry(), 434
id, 32, 91, 308
if, 256
if else, 259
importScripts(), 611
includes(), 284, 287
index(), 484
indexOf(), 284, 288, 289, 291
innerHeight, 302
innerHTML, 318
innerWidth, 302
input, 345
insertAdjacentHTML(), 318, 320
invalid, 350
invert(), 127
isNaN(), 269

J

join(), 289, 291
JSON, 576
justify-content, 180, 182

K

key, 332, 479, 492
key(), 474
KeyboardEvent, 332
keydown, 243
keyPath, 492
keypress, 243
keyup, 243
kind, 371, 375, 383

L

label, 371, 375, 383, 579
language, 375
lastIndexOf(), 284, 288, 289, 291
length, 283, 284, 288, 289, 474, 527, 588
lengthComputable, 504
letter-spacing, 100
lighter, 409
linear-gradient(), 122
linearRampToValueAtTime(), 592
LineBasicMaterial(), 435
lineCap, 402
line-height, 101
lineJoin, 402
lineTo(), 396
lineWidth, 402
listener, 586
list-style, 164
load, 243, 244, 543
load(), 364
loaded, 504
loadend, 504, 543
loadstart, 504, 543
localStorage, 473, 477
location, 302
log(), 337, 338
log10(), 301
lookAt(), 432
loop, 357, 362, 587, 590
low, 72
lowerBound(), 495

M

map(), 289, 294
margin, 109, 312
Math, 300, 400, 421
max, 64, 72
max(), 300
maxDecibels, 602
maxDistance, 599
max-height, 178
maximumAge, 522
maxlength, 62
max-width, 178, 217, 221
measureText(), 404
Mesh(), 433
MeshBasicMaterial(), 435
MeshFaceMaterial(), 436
MeshLambertMaterial(), 435
MeshNormalMaterial(), 435
MeshPhongMaterial(), 435, 445
message, 553, 562, 579, 605
MessageEvent, 556
metaKey, 329, 333
min, 64, 72

min(), 300
minDecibels, 602
min-height, 178
minlength, 62
min-width, 178, 217
miterLimit, 402
mode, 375
mousedown, 242
mouseenter, 242
MouseEvent, 329
mouseleave, 243
mousemove, 243
mouseout, 243
mouseover, 243
mouseup, 242
movementX, 330, 467
movementY, 330, 467
moveTo(), 396
multiple, 76
muted, 357, 363, 385

N

nameservers, 12
navigator, 302
needsUpdate, 444
negotiationneeded, 571
new, 278, 282
newValue, 479
novalidate, 76, 77
Number(), 281
numberOfChannels, 588

O

Object, 279
objectStore(), 483
OctahedronGeometry(), 434
offset, 444
offsetHeight, 314
offsetLeft, 314
offsetTop, 314
offsetWidth, 314
offsetX, 329, 331
offsetY, 329, 331
oldValue, 479
onclick, 242, 323
onkeydown, 243
onkeypress, 243
onkeyup, 243
onload, 243
only(), 494
onmousedown, 242
onmouseenter, 242
onmouseleave, 243
onmousemove, 243
onmouseout, 243
onmouseover, 243
onmouseup, 242
onunload, 243
onwheel, 243
opacity(), 127
open, 562, 579
open(), 303, 305, 481, 539
optimum, 72
order, 179
origin, 553
OrthographicCamera(), 432
outerHTML, 318
outline, 117
outline-color, 117
outline-offset, 117
outline-style, 117
outline-width, 117
overflow, 107, 144, 313
overflow-wrap, 107
overflow-x, 107
overflow-y, 107

P

padding, 109, 312
pageX, 329
pageY, 329
panningModel, 599
parse(), 576
parseFloat(), 269
parseInt(), 269
ParticleBasicMaterial(), 436
pattern, 76, 79
patternMismatch, 351, 353
pause, 364
pause(), 364
paused, 363
perspective, 133
perspective(), 132
PerspectiveCamera(), 432, 438
perspective-origin, 133
PHP, 9, 74
ping, 44
placeholder, 76, 77
PlaneGeometry(), 434
play, 364
play(), 364
pointerlockchange, 464
pointerLockElement, 465, 466

pointerlockerror, 464
PointLight(), 440
pop(), 288, 293
popstate, 531
port, 612
position, 150
POST, 73
poster, 357
postMessage(), 553, 554, 605
pow(), 301
preload, 357, 362
preventDefault(), 325, 512
progress, 364, 504, 543
ProgressEvent, 504
Promise, 381
prompt(), 303
protocol, 561
push(), 288, 292
pushState(), 528, 530
put(), 483
putImageData(), 414
Python, 9

Q

quadraticCurveTo(), 397, 401
querySelector(), 308, 310
querySelectorAll(), 308, 311

R

radial-gradient(), 123, 126
random(), 300, 301
rangeOverflow, 351
rangeUnderflow, 351
readAsArrayBuffer(), 498
readAsBinaryString(), 498
readAsDataURL(), 498
readAsText(), 498
readonly, 76, 484
readwrite, 484
readyState, 487, 561, 579
rect(), 396
refDistance, 599
reliable, 579
reload(), 304
rem, 214, 216
remove(), 316
removeChild(), 321
removeCue(), 378
removeEventListener(), 324
removeItem(), 475
removestream, 571
removeStream(), 571
render(), 430
repeat, 333, 444
replace(), 284, 288, 304
replaceState(), 528
requestAnimationFrame(), 422
requestFullscreen(), 387
requestPointerLock(), 463, 467
required, 76, 78
reset(), 345
response, 542
responseText, 542
responseType, 542
responseXML, 542
restore(), 408
result, 487
return, 267
reverse(), 289, 294
revokeObjectURL(), 503
rgb(), 104
rgba(), 104
rolloffFactor, 599
rotate(), 128, 129, 406
rotate3d(), 132
round(), 300
RTCIceCandidate(), 569
RTCPeerConnection(), 569
Ruby, 9

S

sampleRate, 586, 588
saturate(), 127
save(), 408
scale(), 128, 406
scale3d(), 132
scene(), 430
screenX, 330
screenY, 330
scrollHeight, 314
scrollLeft, 314
scrollTop, 314
scrollWidth, 314
scrollX, 302
scrollY, 303
Selector Universal, 160
send(), 539, 561, 579
sepia(), 127
sessionStorage, 473
setClearColor(), 430
setCustomValidity(), 348
setData(), 508
setDate(), 296

setDragImage(), 508, 514
setFullYear(), 296
setHours(), 296, 299
setInterval(), 303, 305
setItem(), 473
setLocalDescription(), 570
setMilliseconds(), 296
setMinutes(), 296
setMonth(), 296
setOrientation(), 599, 600
setPosition(), 599, 600
setRemoteDescription(), 570
setSeconds(), 296
setSize(), 430
setTargetAtTime(), 593
setTimeout(), 303, 305
setTransform(), 406, 407
setValueAtTime(), 592
setValueCurveAtTime(), 593
setVelocity(), 599, 600
setViewport(), 430
ShaderMaterial(), 436
shadowBlur, 405
shadowColor, 405
shadowOffsetX, 405
shadowOffsetY, 405
SharedWorker(), 612
shift(), 293
shiftKey, 329, 333
sin(), 301
skew(), 128, 130
slice(), 289, 290, 501
smoothingTimeConstant, 602
some(), 291
sort(), 289, 294
source, 487, 553, 556
source-atop, 409
source-in, 409
source-out, 409
speedOfSound, 599
spellcheck, 76
SphereGeometry(), 433
splice(), 288, 293
SpotLight(), 440
sqrt(), 301
src, 102, 357, 361, 370
srclang, 370
srcObject, 381
srcset, 46
start(), 587, 612
startsWith(), 283, 287
state, 528
statechange, 571
step, 64
stepMismatch, 351
stop(), 383, 587
stopPropagation(), 325
storage, 478
storageArea, 479
String(), 281
stringify(), 576
stroke(), 396
strokeRect(), 393
strokeStyle, 394
strokeText(), 403
style, 84
submit(), 345
substr(), 283, 286
substring(), 283, 286
success, 486
switch, 259

T

tan(), 301
target, 325
terminate(), 609
TetrahedronGeometry(), 434
text-align, 100, 101
textAlign, 313, 403
text-align-last, 100
textBaseline, 403
text-decoration, 102
textDecoration, 313
text-indent, 100
text-shadow, 120, 122
textTracks, 374
Texture(), 442
then(), 381
this, 274
throw, 340
timeout, 522, 542, 543
timestamp, 520
toBlob(), 418
toDataURL(), 417
toDateString(), 296, 297
toggle(), 316, 317
toLowerCase(), 283
tooLong, 351
toString(), 296, 297
total, 504
toTimeString(), 296, 297
toUpperCase(), 283, 285
track, 374
transaction, 487
transaction(), 484

transform, 131
transform(), 406, 407
transition, 135
transition-delay, 135
transition-duration, 135
transition-property, 134
transition-timing-function, 135
translate, 55
translate(), 128, 131, 406
translate3d(), 132
trim(), 283, 285
trunc(), 300
try, 340
type, 325
typeMismatch, 351
types, 507

U

unload, 243
unmuted, 385
unshift(), 292
update(), 490
upgradeneeded, 482, 486, 487
upload, 546
upperBound(), 495
url, 479, 561
urls, 573

V

valid, 351
validity, 353
ValidityState, 351
value, 492, 592
valueMissing, 351, 353
versionchange, 484
vertical-align, 101
verticalAlign, 313
viewport, 204
visibility, 140, 231, 312
visibilitychange, 535, 536
visibilityState, 535
volume, 363
VTTCue(), 378

W

watchPosition(), 519, 523
WebGLRenderer(), 430
WebSocket(), 561
WebVTT, 371, 373
wheel, 243
while, 261
window, 527
word-spacing, 100, 101
Worker(), 605, 607

X

XMLHttpRequest(), 539
XMLHttpRequestUpload, 546
xor, 409

Z

z-index, 152
zIndex, 313

El gran libro de HTML5, CSS3 y JavaScript guía al lector paso a paso en el desarrollo de sitios y aplicaciones web. Después de leer este libro sabrá cómo estructurar sus documentos con HTML, cómo otorgarles estilos con CSS y cómo trabajar con las más poderosas APIs de JavaScript.

Este libro es un curso completo que le enseñará cómo construir sitios webs adaptables y aplicaciones revolucionarias desde cero. Cada capítulo explora conceptos básicos y complejos de HTML, CSS y JavaScript. La información viene acompañada por ejemplos funcionales que guían al recién iniciado y también al programador experto a través de cada etiqueta, estilo y función incluidos en estos lenguajes.

Este libro incluye:

Introducción a HTML, CSS y JavaScript | Modelos de Caja Tradicional y Flexible | Diseño Web Adaptable | Vídeo y Audio | API Formularios | API Canvas | API WebGL | API Web Audio | API IndexedDB | API Web Storage | API File | API WebSocket | API WebRTC | API Stream | API Fullscreen | Ajax Level 2 | API Web Workers | API Drag and Drop | API History | API Web Messaging | API Pointer Lock | API Geolocation | API Page Visibility | API TextTrack y más...

Además, en la parte inferior de la primera página del libro encontrará el código de acceso que le permitirá acceder de forma gratuita a los contenidos adicionales del libro en **www.marcombo.info**.

www.marcombo.com