



## 1. Prefacio

- a. [Convenciones utilizadas en este libro](#)
- b. [Código de ejemplo](#)
- c. [Aprendizaje en línea de O'Reilly](#)
- d. [Cómo contactar con nosotros](#)
- e. [Agradecimientos](#)

## 2. Introducción a JavaScript

- a. [1.1 Explorando JavaScript](#)
- b. [1.2 Hola Mundo](#)
- c. [1.3 Un recorrido por JavaScript](#)
- d. [1.4 Ejemplo: Histogramas de frecuencia de caracteres](#)
- e. [1.5 Resumen](#)

## 3. Estructura léxica

- a. [2.1 El texto de un programa JavaScript](#)
- b. [2.2 Observaciones](#)
- c. [2.3 Literales](#)
- d. [2.4 Identificadores y palabras reservadas](#)
  - i. [2.4.1 Palabras reservadas](#)
- e. [2.5 Unicode](#)

- i. 2.5.1 Secuencias de Escape Unicode
  - ii. 2.5.2 Normalización Unicode
  - f. 2.6 Punto y coma opcional
  - g. 2.7 Resumen
4. Tipos, valores y variables
- a. 3.1 Visión general y definiciones
  - b. 3.2 Números
    - i. 3.2.1 Literales enteros ii. 3.2.2 Literales en coma flotante iii. 3.2.3 Aritmética en JavaScript
    - iv. 3.2.4 Errores de punto flotante binario y de redondeo
    - v. 3.2.5 Números enteros de precisión arbitraria con BigInt
    - vi. 3.2.6 Fechas y horarios
  - c. 3.3 Texto
    - i. 3.3.1 Literales de cadena ii. 3.3.2 Secuencias de escape en literales de cadena iii. 3.3.3 Trabajo con cadenas iv. 3.3.4 Literales de plantilla
    - v. 3.3.5 Comparación de patrones
  - d. 3.4 Valores booleanos

- e. 3.5 nulos e indefinidos
  - f. 3.6 Símbolos
  - g. 3.7 El objeto global
  - h. 3.8 Valores primitivos inmutables y referencias a objetos mutables
  - i. 3.9 Conversiones de tipo
    - i. 3.9.1 Conversiones e igualdades
    - ii. 3.9.2 Conversiones explícitas
    - iii. 3.9.3 Conversiones de objetos a primitivos
  - j. 3.10 Declaración y asignación de variables
    - i. 3.10.1 Declaraciones con let y const
    - ii. 3.10.2 Declaraciones de variables con var
    - iii. 3.10.3 Asignación de la desestructuración
  - k. 3.11 Resumen
5. Expresiones y operadores
- a. 4.1 Expresiones primarias
  - b. 4.2 Inicializadores de objetos y matrices
  - c. 4.3 Expresiones de definición de funciones
  - d. 4.4 Expresiones de acceso a la propiedad
    - i. 4.4.1 Acceso condicional a la propiedad

- e. 4.5 Expresiones de invocación
  - i. 4.5.1 Invocación condicional
- f. 4.6 Expresiones de creación de objetos
- g. 4.7 Visión general del operador
  - i. 4.7.1 Número de operandos

- ..
- ii. 4.7.2 Tipo de operando y resultado iii. 4.7.3 Efectos secundarios del operador iv. 4.7.4 Precedencia del operador
  - v. 4.7.5 Asociatividad de los operadores
  - vi. 4.7.6 Orden de evaluación
  - h. 4.8 Expresiones aritméticas
  - i. 4.8.1 El operador + ii. 4.8.2 Operadores aritméticos unarios
  - iii. 4.8.3 Operadores Bitwise
  - i. 4.9 Expresiones relacionales
    - i. 4.9.1 Operadores de igualdad y desigualdad ii. 4.9.2 Operadores de comparación iii. 4.9.3 El operador in
    - iv. 4.9.4 El operador instanceof
  - j. 4.10 Expresiones lógicas
    - i. 4.10.1 AND lógico (&&) ii. 4.10.2 OR lógico (||)
    - iii. 4.10.3 Lógica NOT (!)
  - k. 4.11 Expresiones de asignación
    - i. 4.11.1 Asignación con operación
  - l. 4.12 Expresiones de evaluación

..

- i. [4.12.1 eval\(\)](#)
- ii. [4.12.2 Global eval\(\)](#)
- iii. [4.12.3 Evaluación estricta](#)

m. [4.13 Operadores diversos](#)

- i. [4.13.1 El operador condicional \(?:\)](#)
- ii. [4.13.2 Definido por primera vez \(??\)](#)
- iii. [4.13.3 El operador typeof](#) iv. [4.13.4 El operador delete](#)
- v. [4.13.5 El operador await](#) vi. [4.13.6 El operador void](#)
- vii. [4.13.7 El operador coma \(,\)](#)

n. [4.14 Resumen](#)

6. [Declaraciones](#)

- a. [5.1 Declaraciones de expresión](#)
- b. [5.2 Declaraciones compuestas y vacías](#)
- c. [5.3 Condicionales](#)
  - i. [5.3.1 si](#) ii. [5.3.2 si no](#)
  - iii. [5.3.3 interruptor](#)
- d. [5.4 Bucles](#)
  - i. [5.4.1 while](#) ii. [5.4.2 do/while](#)

..

iii. 5.4.3 para

iv. 5.4.4 para/de

v. 5.4.5 para/in

e. 5.5 Saltos

i. 5.5.1 Declaraciones

etiquetadas ii. 5.5.2 break iii.

5.5.3 continue iv. 5.5.4

return

v. 5.5.5 ceder

vi. 5.5.6 tirar

vii. 5.5.7 try/catch/finally

f. 5.6 Declaraciones diversas

i. 5.6.1 con ii.

5.6.2 depurador

iii. 5.6.3 "uso estricto"

g. 5.7 Declaraciones

i. 5.7.1 const, let y var ii.

5.7.2 función iii. 5.7.3 clase

iv. 5.7.4 importación y exportación

h. 5.8 Resumen de las declaraciones de  
JavaScript

7. Objetos

..

- a. [6.1 Introducción a los objetos](#)
- b. [6.2 Crear objetos](#)
  - i. [6.2.1 Literales de objetos](#) ii. [6.2.2 Creación de objetos con nuevos](#) iii. [6.2.3 Prototipos](#)
  - iv. [6.2.4 Object.create\(\)](#)
- c. [6.3 Consulta y configuración de propiedades](#)
  - i. [6.3.1 Objetos como matrices asociativas](#) ii. [6.3.2 Herencia](#)
  - iii. [6.3.3 Errores de acceso a la propiedad](#)
- d. [6.4 Borrar propiedades](#)
- e. [6.5 Comprobación de las propiedades](#)
- f. [6.6 Enumeración de propiedades](#)
  - i. [6.6.1 Orden de enumeración de las propiedades](#)
- g. [6.7 Ampliación de objetos](#)
- h. [6.8 Serialización de objetos](#)
- i. [6.9 Métodos de los objetos](#)
  - i. [6.9.1 El método toString\(\)](#) ii. [6.9.2 El método toLocaleString\(\)](#) iii. [6.9.3 El método valueOf\(\)](#)

..  
iv. 6.9.4 El método toJSON() j.

## 6.10 Sintaxis literal de objetos extendida

- i. 6.10.1 Propiedades abreviadas
  - ii. 6.10.2 Nombres de propiedades computados
  - iii. 6.10.3 Símbolos como nombres de propiedades
  - iv. 6.10.4 Operador de propagación
  - v. 6.10.5 Métodos abreviados
  - vi. 6.10.6 Obtención y fijación de propiedades
- k. 6.11 Resumen

## 8. Arrays

- a. 7.1 Creación de matrices
  - i. 7.1.1 Literales de arrays
  - ii. 7.1.2 El operador Spread
  - iii. 7.1.3 El constructor Array()
  - iv. 7.1.4 Array.of()
  - v. 7.1.5 Array.from()
- b. 7.2 Lectura y escritura de elementos de la matriz
- c. 7.3 Matrices dispersas
- d. 7.4 Longitud de la matriz

..

- e. [7.5 Añadir y eliminar elementos de la matriz](#)
- f. [7.6 Iteración de matrices](#)
- g. [7.7 Matrices multidimensionales](#)
- h. [7.8 Métodos de la matriz](#)
- i. [7.8.1 Métodos de Iteración de Matrices](#)
  - ii. [7.8.2 Aplanar matrices con flat\(\) y flatMap\(\)](#)

- iii. [7.8.3 Añadir matrices con concat\(\)](#)
- iv. [7.8.4 Pilas y colas con push\(\), pop\(\), shift\(\) y unshift\(\)](#)
- v. [7.8.5 Submatriz con slice\(\), splice\(\), fill\(\) y copyWithin\(\)](#)
- vi. [7.8.6 Métodos de búsqueda y ordenación de matrices](#)
- vii. [7.8.7 Conversiones de matrices a cadenas](#)
- viii. [7.8.8 Funciones estáticas de las matrices](#)
  - i. [7.9 Objetos similares a matrices](#)
  - j. [7.10 Cadenas como matrices](#)
  - k. [7.11 Resumen](#)

## 9. [Funciones](#)

- a. [8.1 Definición de funciones](#)
  - i. [8.1.1 Declaraciones de funciones](#) ii. [8.1.2 Expresiones de funciones](#) iii. [8.1.3 Funciones de flecha](#)
  - iv. [8.1.4 Funciones anidadas](#)
- b. [8.2 Invocación de funciones](#)

i. 8.2.1 Invocación de funciones

ii. 8.2.2 Invocación de métodos

iii. 8.2.3 Invocación de

constructores iv. 8.2.4

Invocación indirecta

v8.2.5 Invocación implícita de funciones

c. 8.3 Argumentos y parámetros de las  
funciones

i. 8.3.1 Parámetros opcionales y  
valores por defecto

ii. 8.3.2 Parámetros restantes y listas  
de argumentos de longitud variable iii.

8.3.3 El objeto Arguments

iv. 8.3.4 El operador Spread para las  
llamadas a funciones

v. 8.3.5 Argumentos de la función de  
desestructuración  
en Parámetros

vi. 8.3.6 Tipos de argumentos

d. 8.4 Funciones como valores

i. 8.4.1 Definir sus propias  
propiedades de función

e. 8.5 Funciones como espacios de nombres

- f. 8.6 Cierres
- g. 8.7 Propiedades, métodos y funciones  
Constructor
  - i. 8.7.1 La propiedad length ii. 8.7.2 La propiedad name iii. 8.7.3 La propiedad prototype iv. 8.7.4 Los métodos call() y apply()
  - v. 8.7.5 El método bind() vi. 8.7.6 El método toString() vii. 8.7.7 El constructor Function()
- h. 8.8 Programación funcional
  - i. 8.8.1 Procesamiento de matrices con funciones ii. 8.8.2 Funciones de orden superior iii. 8.8.3 Aplicación parcial de funciones
  - iv. 8.8.4 Memoización
- i. 8.9 Resumen

## 10. Clases

- a. 9.1 Clases y prototipos
- b. 9.2 Clases y Constructores
  - i. 9.2.1 Constructores, identidad de clase y

de

ii. 9.2.2 La propiedad del constructor

c. 9.3 Clases con la palabra clave class

i. 9.3.1 Métodos estáticos

ii. 9.3.2 Getters, Setters y otras formas de métodos

iii. 9.3.3 Campos públicos, privados y estáticos

iv. 9.3.4 Ejemplo: Una clase de números complejos

d. 9.4 Añadir métodos a las clases existentes

e. 9.5 Subclases

i. 9.5.1 Subclases y prototipos ii. 9.5.2

Subclases con extends y super iii. 9.5.3

Delegación en lugar de herencia

iv. 9.5.4 Jerarquías de clases y clases abstractas

f. 9.6 Resumen

11. Módulos

a. 10.1 Módulos con clases, objetos y cierres

i. 10.1.1 Automatización de la modularidad basada en el cierre

b. 10.2 Módulos en el nodo

i. [10.2.1 Exportación de](#)

nodos ii. [10.2.2](#)

Importación de nodos

iii. [10.2.3 Módulos tipo nodo en la web](#)

c. [10.3 Módulos en ES6](#)

i. [10.3.1 Exportaciones](#)

ES6 ii. [10.3.2](#)

Importaciones ES6

iii. [10.3.3 Importaciones y exportaciones con cambio de nombre](#)

iv. [10.3.4 Reexportaciones](#)

v. [10.3.5 Módulos JavaScript en la Web](#) vi.

[10.3.6 Importaciones dinámicas con import\(\)](#)

vii. [10.3.7 importar.meta.url](#)

d. [10.4 Resumen](#)

12. [La biblioteca estándar de JavaScript](#)

- a. 11.1 Conjuntos y mapas
  - i. 11.1.1 La clase Set ii. 11.1.2 La clase Map
  - iii. 11.1.3 WeakMap y WeakSet
- b. 11.2 Matrices tipificadas y datos binarios
  - i. 11.2.1 Tipos de matrices tipificadas ii. 11.2.2 Creación de matrices tipificadas iii. 11.2.3 Uso de matrices tipificadas
  - iv. 11.2.4 Métodos y propiedades de matrices tipificadas
  - v. 11.2.5 DataView y Endianness
- c. 11.3 Comparación de patrones con expresiones regulares
  - i. 11.3.1 Definición de expresiones regulares
  - ii. 11.3.2 Métodos de cadenas para la comparación de patrones
  - iii. 11.3.3 La clase RegExp
- d. 11.4 Fechas y horarios
  - i. 11.4.1 Marcas de tiempo ii. 11.4.2 Aritmética de fechas
  - iii. 11.4.3 Formato y análisis de cadenas de fechas

- e. [11.5 Clases de error](#)
  - f. [11.6 Serialización y análisis de JSON](#)
    - i. [11.6.1 Personalizaciones JSON](#)
  - g. [11.7 La API de internacionalización](#)
    - i. [11.7.1 Formato de números](#) ii.  
[11.7.2 Formato de fechas y horas](#)
    - iii. [11.7.3 Comparación de cadenas](#)
  - h. [11.8 La API de la consola](#)
    - i. [11.8.1 Salida formateada con la consola](#)
    - i. [11.9 APIs de URLs](#)
      - i. [11.9.1 Funciones de URL heredadas](#)
  - j. [11.10 Temporizadores](#)
  - k. [11.11 Resumen](#)
13. [Iteradores y generadores](#)
- a. [12.1 Cómo funcionan los iteradores](#)
  - b. [12.2 Implementación de objetos iterables](#)
    - i. [12.2.1 "Cerrar" un Iterador: El retorno Método](#)
  - c. [12.3 Generadores](#)
    - i. [12.3.1 Ejemplos de generadores](#)
    - ii. [12.3.2 Generadores de rendimiento\\* y recursivos](#)

- d. [12.4 Características avanzadas del generador](#)
  - i. [12.4.1 El valor de retorno de una función generadora](#)
  - ii. [12.4.2 El valor de una expresión de rendimiento](#)
  - iii. [12.4.3 Los métodos return\(\) y throw\(\) de un generador](#)
  - iv. [12.4.4 Una nota final sobre los generadores](#)

- e. [12.5 Resumen](#)

- 14. [JavaScript asíncrono](#)
  - a. [13.1 Programación asíncrona con callbacks](#)
    - i. [13.1.1 Temporizadores](#)
    - ii. [13.1.2 Eventos](#) iii. [13.1.3 Eventos de red](#)
    - iv. [13.1.4. Llamadas de retorno y eventos en Node](#)
  - b. [13.2 Promesas](#)
    - i. [13.2.1 Uso de promesas](#) ii. [13.2.2 Encadenamiento de promesas](#) iii. [13.2.3 Resolución de promesas](#) iv. [13.2.4 Más sobre promesas y errores](#)
    - v. [13.2.5 Promesas en paralelo](#)
    - vi. [13.2.6 Hacer promesas](#)

vii. [13.2.7 Promesas en secuencia](#)

c. [13.3 async y await](#)

i. [13.3.1 Expresiones await](#) ii. [13.3.2 Funciones async](#)

iii. [13.3.3 Esperando múltiples promesas](#) iv. [13.3.4 Detalles de implementación](#)

d. 13.4 Iteración asíncrona

i. 13.4.1 El bucle for/await ii.

13.4.2 Iteradores asíncronos iii.

13.4.3 Generadores asíncronos

iv. 13.4.4. Implementación de Iteradores Asíncronos

e. 13.5 Resumen

15. Metaprogramación

a. 14.1 Atributos de las propiedades

b. 14.2 Extensibilidad de los objetos

c. 14.3 El atributo prototipo

d. 14.4 Símbolos conocidos

i. 14.4.1 Symbol.iterator y

Symbol.asyncIterator ii. 14.4.2

Symbol.hasInstance iii. 14.4.3

Symbol.toStringTag iv. 14.4.4

Symbol.species

v. 14.4.5 Symbol.isConcatSpreadable

vi. 14.4.6 Símbolos de coincidencia de patrones vii. 14.4.7 Symbol.toPrimitive

viii. 14.4.8 Símbolo.unscopable

e. 14.5 Etiquetas de las plantillas

f. 14.6 La API de Reflect

g. 14.7 Objetos Proxy

i. 14.7.1 Invariantes de los proxies

h. 14.8 Resumen

16. JavaScript en los navegadores web

a. 15.1 Fundamentos de la programación web

i. 15.1.1 JavaScript en las etiquetas HTML

<script> ii. 15.1.2 El modelo de objetos del documento

iii. 15.1.3 El objeto global en los navegadores web

iv. 15.1.4 Los scripts comparten un espacio de nombres

v. 15.1.5 Ejecución de programas

JavaScript vi. 15.1.6 Entrada y salida del

programa vii. 15.1.7 Errores del programa

viii. 15.1.8 El modelo de seguridad web

b. 15.2 Eventos

i. 15.2.1 Categorías de eventos ii.

15.2.2 Registro de manejadores de

eventos iii. 15.2.3 Invocación de

manejadores de eventos iv. 15.2.4

Propagación de Eventos

v. 15.2.5 Cancelación de eventos

vi. 15.2.6 Envío de eventos personalizados

c. 15.3 Documentos de scripting

i. 15.3.1 Selección de elementos del documento ii. 15.3.2 Estructura del documento y recorrido iii. 15.3.3 Atributos iv.

15.3.4 Contenido del elemento

v. 15.3.5 Crear, insertar y eliminar nodos

vi. 15.3.6 Ejemplo: Generación de un índice de contenidos

d. 15.4 Scripting CSS

i. 15.4.1 Clases CSS ii. 15.4.2

Estilos en línea iii. 15.4.3

Estilos calculados iv. 15.4.4

Hojas de estilo con script

v. 15.4.5 Animaciones y eventos CSS

e. 15.5 Geometría del documento y desplazamiento

i. 15.5.1 Coordenadas del documento y coordenadas de la ventana gráfica

ii. 15.5.2 Consultar la geometría de un elemento

iii. 15.5.3 Determinación del elemento en un punto

- iv. 15.5.4 Desplazamiento
- v. 15.5.5 Tamaño de la ventana gráfica, tamaño del contenido y Posición de desplazamiento
- f. 15.6 Componentes web
  - i. 15.6.1 Uso de componentes web
  - ii. 15.6.2 Plantillas HTML
  - iii. 15.6.3 Elementos personalizados
  - iv. 15.6.4 Shadow DOM
  - v. 15.6.5 Ejemplo: un componente web <search-box>
- g. 15.7 SVG: gráficos vectoriales escalables
  - i. 15.7.1 SVG en HTML
  - ii. 15.7.2 Scripting SVG
  - iii. 15.7.3 Creación de imágenes SVG con JavaScript
- h. 15.8 Gráficos en un <lienzo>
  - i. 15.8.1 Caminos y polígonos
  - ii. 15.8.2 Dimensiones y coordenadas del lienzo
  - iii. 15.8.3 Atributos de los gráficos
  - iv. 15.8.4 Operaciones de dibujo en el lienzo

- v. 15.8.5 Transformación del sistema de coordenadas vi. 15.8.6 Recorte
- vii. 15.8.7 Manipulación de píxeles
- i. 15.9 APIs de audio
  - i. 15.9.1 El constructor Audio()
  - ii. 15.9.2 La API WebAudio
- j. 15.10 Ubicación, navegación e historia
  - i. 15.10.1 Carga de nuevos documentos ii. 15.10.2 Historial de navegación
  - iii. 15.10.3 Gestión del historial con eventos de intercambio de hash
  - iv. 15.10.4 Gestión del historial con pushState()
- k. 15.11 Creación de redes
  - i. 15.11.1 fetch() ii. 15.11.2 Eventos enviados por el servidor
  - iii. 15.11.3 WebSockets
- l. 15.12 Almacenamiento
  - i. 15.12.1 localStorage y sessionStorage ii. 15.12.2 Cookies
  - iii. 15.12.3 IndexedDB
- m. 15.13 Hilos de trabajo y mensajería

i. 15.13.1 Objetos del trabajador ii.

15.13.2 El objeto global en los trabajadores iii. 15.13.3 Importación de código en un trabajador iv. 15.13.4 Modelo de ejecución del trabajador

v. 15.13.5 postMessage(), MessagePorts, y MessageChannels

- vi. [15.13.6. Mensajería cruzada con postMessage\(\)](#)
- n. [15.14 Ejemplo: El conjunto de Mandelbrot](#)
- o. [15.15 Resumen y sugerencias para el futuro Leer](#)
  - i. [15.15.1 HTML y CSS](#) ii. [15.15.2 Rendimiento](#) iii. [15.15.3 Seguridad](#)
  - iv. [15.15.4 WebAssembly](#)
  - v. [15.15.5 Más funciones de documentos y ventanas](#) vi. [15.15.6 Eventos](#)
  - vii. [15.15.7 Aplicaciones web progresivas y Service Workers](#) viii. [15.15.8 APIs para dispositivos móviles](#)
  - ix. [15.15.9 APIs binarias](#)
  - x. [15.15.10 APIs de medios de comunicación](#)
  - xi. [15.15.11 Criptografía y APIs relacionadas](#)

## 17. JavaScript del lado del servidor con Node

- a. [16.1 Fundamentos de la programación de nodos](#)
  - i. [16.1.1 Salida de la consola](#)

- ii. 16.1.2 Argumentos de la línea de comandos y variables de entorno
- iii. 16.1.3 Ciclo de vida del programa
- v. 16.1.4 Módulos de nodos
- vi. 16.1.5 El gestor de paquetes de nodos
- b. 16.2 El nodo es asíncrono por defecto
- c. 16.3 Bufferes
- d. 16.4 Eventos y EventEmitter
- e. 16.5 Arroyos
  - i. 16.5.1 Tuberías ii. 16.5.2 Iteración asíncrona
  - iii. 16.5.3 Escribir en flujos y manejar la contrapresión
  - iv. 16.5.4 Lectura de flujos con eventos
- f. 16.6 Detalles del proceso, la CPU y el sistema operativo
- g. 16.7 Trabajar con archivos
  - i. 16.7.1 Rutas, descriptores de archivos y FileHandles
  - ii. 16.7.2 Lectura de archivos
  - iii. 16.7.3 Escritura de archivos

- iv. [16.7.4 Operaciones con archivos](#)
- v. [16.7.5 Metadatos de los archivos](#)
- vi. [16.7.6 Trabajar con directorios](#)
- h. [16.8 Clientes y servidores HTTP](#)
- i. [16.9 Servidores y clientes de red no HTTP](#)
- j. [16.10 Trabajar con procesos infantiles](#)
  - i [16.10.1 execSync\(\) y execFileSync\(\)](#)
  - ii. [16.10.2 exec\(\) y execFile\(\)](#)
  - iii. [16.10.3 spawn\(\)](#)
  - iv. [16.10.4 fork\(\)](#)
- k. [16.11 Hilos de trabajo](#)
  - i. [16.11.1 Crear trabajadores y pasar mensajes](#)
  - ii. [16.11.2 El entorno de ejecución del trabajador](#)
  - iii. [16.11.3 Canales de comunicación y MessagePorts](#)
  - iv. [16.11.4. Transferencia de MessagePorts y Arrays tipificados](#)
  - v. [16.11.5. Compartir arrays tipificados entre hilos](#)
- l. [16.12 Resumen](#)

## 18. Herramientas y extensiones de JavaScript

- a. 17.1. La limpieza con ESLint
- b. 17.2 Formato de JavaScript con Prettier
- c. 17.3 Pruebas unitarias con Jest
- d. 17.4 Gestión de paquetes con npm
- e. 17.5 Agrupación de códigos
- f. 17.6 Transpilación con Babel
- g. 17.7 JSX: Expresiones de marcado en JavaScript

- h. [17.8 Comprobación de tipo con flujo](#)
- i. [17.8.1 Instalación y ejecución de Flow](#) ii. [17.8.2](#)

Uso de anotaciones de tipo iii. [17.8.3 Tipos de clase](#) iv.

#### 17.8.4 Tipos de objeto

v. [17.8.5 Alias de tipos](#) vi. [17.8.6](#)

Tipos de matrices vii. [17.8.7 Otros](#)

tipos parametrizados viii. [17.8.8](#)

Tipos de sólo lectura ix. [17.8.9 Tipos de función](#)

x. [17.8.10 Tipos de unión](#)

xi. [17.8.11. Tipos enumerados y uniones discriminadas](#)

i. [17.9 Resumen](#)

## 19. Índice

### **Elogios para *JavaScript: La guía definitiva*, séptima edición**

*"Este libro es todo lo que nunca supo que quería saber sobre JavaScript. Lleve la calidad y la productividad de su código JavaScript al siguiente nivel. El conocimiento de David sobre el lenguaje, sus complejidades y sus trucos, es asombroso, y brilla en esta guía verdaderamente definitiva del lenguaje JavaScript."*

-Schalk Neethling, Ingeniero Senior de Frontend en  
Documentos web de  
MDN

*"David Flanagan lleva a los lectores a una visita guiada por JavaScript que les proporcionará una imagen completa del lenguaje y su ecosistema".*

-Sarah Wachs, Desarrolladora Frontend y Mujer que  
Código de Berlín Jefe

*"Cualquier desarrollador interesado en ser productivo en las bases de código desarrolladas a lo largo de la vida de JavaScript (incluyendo las características más recientes y emergentes) estará bien servido por un viaje profundo y reflexivo a través de este libro completo y definitivo."*

-Brian Sletten, Presidente de Bosatsu Consulting

# JavaScript: La Guía Definitiva

SÉPTIMA EDICIÓN

Domine el lenguaje de programación más  
utilizado del mundo

**David Flanagan**



**JavaScript: The Definitive Guide, Seventh Edition** de  
David Flanagan

Copyright © 2020 David Flanagan. Todos los derechos reservados.

Impreso en los Estados Unidos de América.

Publicado por O'Reilly Media, Inc., 1005 Gravenstein Highway North,  
Sebastopol, CA 95472.

Los libros de O'Reilly pueden adquirirse para uso educativo,  
comercial o promocional. La mayoría de los títulos también están  
disponibles en línea (<http://oreilly.com>). Para más información,

póngase en contacto con nuestro departamento de ventas para empresas e instituciones: 800-998-9938 o [corporate@oreilly.com](mailto:corporate@oreilly.com).

Editora de adquisiciones: Jennifer Pollock

Editor de desarrollo: Angela Rufino

Editor de producción: Deborah Baker

Redactor: Holly Bauer Forsyth

Corrector de pruebas: Piper Editorial, LLC

Indexador: Judith McConville

Diseñador de interiores: David Futato

Diseñador de la cubierta: Karen Montgomery

Ilustrador: Rebecca Demarest

Junio de 1998: Tercera edición

Noviembre de 2001: cuarta edición

Agosto de 2006: Quinta edición

Mayo de 2011: Sexta edición

Mayo de 2020: Séptima edición

### **Historial de revisiones de la séptima edición**

- 2020-05-13: Primera publicación

Consulte

<http://oreilly.com/catalog/errata.csp?isbn=9781491952023> para conocer los detalles del lanzamiento.

El logotipo de O'Reilly es una marca registrada de O'Reilly Media, Inc. *JavaScript: The Definitive Guide*, Seventh Edition, la imagen de la portada y la imagen comercial relacionada son marcas comerciales de O'Reilly Media, Inc.

Aunque el editor y los autores se han esforzado de buena fe para garantizar que la información y las instrucciones contenidas en esta obra sean precisas, el editor y los autores declinan toda responsabilidad por errores u omisiones, incluyendo, sin limitación, la responsabilidad por los daños resultantes del uso o la confianza en esta obra. El uso de la información y las instrucciones contenidas en esta obra es bajo su propio riesgo. Si cualquier muestra de código u otra tecnología que esta obra contenga o describa está sujeta a licencias de código abierto o a derechos de propiedad intelectual de terceros, es su responsabilidad asegurarse de que su uso cumple con dichas licencias y/o derechos.

978-1-491-95202-3

[LSI]

## **Dedicación**

A mis padres, Donna y Matt, con amor y gratitud.

# Prefacio

---

Este libro cubre el lenguaje JavaScript y las APIs de JavaScript implementadas por los navegadores web y por Node. Lo he escrito para lectores con alguna experiencia previa en programación que quieran aprender JavaScript y también para programadores que ya usan JavaScript pero quieren llevar su comprensión a un nuevo nivel y dominar realmente el lenguaje. Mi objetivo con este libro es documentar el lenguaje JavaScript de forma exhaustiva y definitiva y proporcionar una introducción en profundidad a las API más importantes del lado del cliente y del lado del servidor disponibles para los programas JavaScript. Por ello, este es un libro largo y detallado. Sin embargo, espero que recompense un estudio cuidadoso y que el tiempo que se invierta en su lectura se recupere fácilmente en forma de una mayor productividad en la programación.

Las ediciones anteriores de este libro incluían una amplia sección de referencias. Ya no creo que tenga sentido incluir ese material en forma impresa cuando es tan rápido y fácil encontrar material de referencia actualizado en línea. Si necesitas buscar algo relacionado con el núcleo o el lado del cliente de JavaScript, te recomiendo que visites el [sitio web de MDN](#). Y para las APIs de Node del lado del servidor, te recomiendo que vayas directamente a la fuente y consultes la [documentación de referencia de Node.js](#).

# Convenciones utilizadas en este libro

En este libro uso las siguientes convenciones tipográficas:

## *Cursiva*

Se utiliza para enfatizar y para indicar el primer uso de un término. *La cursiva* también se utiliza para las direcciones de correo electrónico, las URL y los nombres de archivos.

## *Anchura constante*

Se utiliza en todo el código JavaScript y en los listados CSS y HTML, y en general para cualquier cosa que se escriba literalmente al programar.

## *Cursiva de ancho constante*

Se utiliza ocasionalmente para explicar la sintaxis de JavaScript.

## **Negrita de ancho constante**

Muestra comandos u otros textos que deben ser escritos literalmente por el usuario

### NOTA

Este elemento significa una nota general.

### IMPORTANTE

Este elemento indica una advertencia o precaución.

## Código de ejemplo

El material complementario (ejemplos de código, ejercicios, etc.) de este libro está disponible para su descarga en

[https://oreil.ly/javascript\\_defgd7](https://oreil.ly/javascript_defgd7)

Este libro está aquí para ayudarle a realizar su trabajo. En general, si el código de ejemplo se ofrece con este libro, puede utilizarlo en sus programas y documentación. No es necesario que se ponga en contacto con nosotros para pedir permiso, a menos que reproduzca una parte importante del código. Por ejemplo, escribir un programa que utilice varios trozos de código de este libro no requiere permiso. Vender o distribuir ejemplos de los libros de O'Reilly sí requiere permiso. Responder a una pregunta citando este libro y citando el código de ejemplo no requiere permiso.

La incorporación de una cantidad significativa de código de ejemplo de este libro en la documentación de su producto requiere autorización.

Agradecemos, pero generalmente no exigimos, la atribución. La atribución suele incluir el título, el autor, la editorial y el ISBN. Por ejemplo: "*JavaScript: The Definitive Guide*, Seventh Edition, por David Flanagan (O'Reilly). Copyright 2020 David Flanagan, 978-1-491-95202-3."

Si cree que el uso de los ejemplos de código no se ajusta al uso legítimo o a los permisos mencionados, no dude en ponerse en contacto con nosotros en [permissions@oreilly.com](mailto:permissions@oreilly.com).

# Aprendizaje en línea de O'Reilly

## NOTA

Durante más de 40 años, *O'Reilly Media* ha proporcionado formación en tecnología y negocios, conocimientos y perspectivas para ayudar a las empresas a tener éxito.

Nuestra red única de expertos e innovadores comparten sus conocimientos y experiencia a través de libros, artículos y nuestra plataforma de aprendizaje en línea. La plataforma de aprendizaje en línea de O'Reilly le permite acceder a cursos de formación en directo, rutas de aprendizaje en profundidad, entornos de codificación interactivos y una amplia colección de textos y videos de O'Reilly y de más de 200 editores. Para más información, visite <http://oreilly.com>.

## Cómo contactar con nosotros

Por favor, dirija sus comentarios y preguntas sobre este libro al editor:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (en Estados Unidos o Canadá)

707-829-0515 (internacional o local)

707-829-0104 (fax)

Disponemos de una página web para este libro, en la que figuran las erratas, los ejemplos y cualquier información adicional. Puede acceder a esta página en [https://oreil.ly/javascript\\_defgd7](https://oreil.ly/javascript_defgd7).

Envíe un correo electrónico [\*a bookquestions@oreilly.com\*](mailto:bookquestions@oreilly.com) para comentar o hacer preguntas técnicas sobre este libro.

Para obtener noticias y más información sobre nuestros libros y cursos, consulte nuestro sitio web en <http://www.oreilly.com>.

Encuéntrenos en Facebook: <http://facebook.com/oreilly>

Síganos en Twitter: <http://twitter.com/oreillymedia>

Míranos en YouTube: <http://www.youtube.com/oreillymedia>

## Agradecimientos

Muchas personas han colaborado en la creación de este libro. Me gustaría dar las gracias a mi editora, Ángela Rufino, por mantenerme en el camino y por su paciencia con el incumplimiento de los plazos. Gracias también a mis revisores técnicos: Brian Sletten, Elisabeth Robson, Ethan Flanagan, Maximiliano Firtman, Sarah Wachs y Schalk Neethling. Sus comentarios y sugerencias han hecho que este libro sea mejor.

El equipo de producción de O'Reilly ha hecho su habitual buen trabajo: Kristen Brown dirigió el proceso de producción, Deborah Baker fue la editora de producción, Rebecca Demarest dibujó las figuras y Judy McConville creó el índice.

Entre los editores, revisores y colaboradores de ediciones anteriores de este libro se encuentran: Andrew Schulman, Angelo Sirigos, Aristóteles Pagaltzis, Brendan Eich, Christian Heilmann, Dan Shafer, Dave C. Mitchell, Deb Cameron, Douglas Crockford, Dr. Tankred Hirschmann, Dylan Schiemann, Frank Willison, Geoff Stearns, Herman Venter, Jay Hodges, Jeff Yates, Joseph Kesselman, Ken Cooper, Larry Sullivan, Lynn Rollins, Neil Berkman, Mike Loukides, Nick Thompson, Norris Boyd, Paula Ferguson, Peter-Paul Koch, Philippe Le Hegaret, Raffaele Cecco, Richard Yaker, Sanders Kleinfeld, Scott Furman, Scott Isaacs, Shon Katzenberger, Terry Allen, Todd Ditchendorf, Vidur Apparao, Waldemar Horwat y Zachary Kessin.

Escribir esta séptima edición me ha mantenido alejado de mi familia durante muchas noches. Mi cariño para ellos y mi agradecimiento por aguantar mis ausencias.

*David Flanagan, marzo de 2020*



# Capítulo 1. Introducción a JavaScript

---

JavaScript es el lenguaje de programación de la web. La inmensa mayoría de los sitios web utilizan JavaScript, y todos los navegadores web modernos -en ordenadores de sobremesa, tabletas y teléfonos- incluyen intérpretes de JavaScript, lo que convierte a este lenguaje en el más utilizado de la historia. En la última década, Node.js ha permitido la programación en JavaScript fuera de los navegadores web, y el espectacular éxito de Node significa que JavaScript es ahora también el lenguaje de programación más utilizado entre los desarrolladores de software. Tanto si empiezas desde cero como si ya utilizas JavaScript de forma profesional, este libro te ayudará a dominar el lenguaje.

Si ya está familiarizado con otros lenguajes de programación, puede ayudarle a saber que JavaScript es un lenguaje de programación interpretado, dinámico y de alto nivel que se adapta bien a los estilos de programación funcional y orientada a objetos. Las variables de JavaScript no están tipadas. Su sintaxis se basa en Java, pero los lenguajes no están relacionados. JavaScript deriva sus funciones de primera clase de Scheme y su herencia basada en prototipos del poco conocido lenguaje Self. Pero no es necesario conocer ninguno de esos lenguajes, ni estar familiarizado con esos términos, para utilizar este libro y aprender JavaScript.

El nombre "JavaScript" es bastante engañoso. Excepto por un parecido sintáctico superficial, JavaScript es completamente diferente de Java

### JAVASCRIPT: NOMBRES, VERSIONES Y MODOS

JavaScript se creó en Netscape en los primeros días de la web y, técnicamente, "JavaScript" es una marca comercial con licencia de Sun Microsystems (ahora Oracle) que se utiliza para describir la implementación del lenguaje de Netscape (ahora Mozilla). Netscape presentó el lenguaje para su estandarización a la ECMA -la Asociación Europea de Fabricantes de Ordenadores- y, por cuestiones de marca, la versión estandarizada del lenguaje se quedó con el incómodo nombre de "ECMAScript". En la práctica, todo el mundo llama al lenguaje JavaScript. Este libro utiliza el nombre "ECMAScript" y la abreviatura "ES" para referirse al estándar del lenguaje y a las versiones de ese estándar.

Durante la mayor parte de la década de 2010, la versión 5 del estándar ECMAScript ha sido compatible con todos los navegadores web. Este libro trata a ES5 como la línea de base de compatibilidad y ya no discute las versiones anteriores del lenguaje. ES6 se publicó en 2015 y añadió nuevas e importantes características -incluyendo la sintaxis de clases y módulos- que hicieron que JavaScript pasara de ser un lenguaje de scripting a un lenguaje serio y de propósito general adecuado para la ingeniería de software a gran escala. Desde ES6, la especificación ECMAScript ha pasado a tener una cadencia de publicación anual, y las versiones del lenguaje -ES2016, ES2017, ES2018, ES2019 y ES2020- se identifican ahora por el año de publicación.

A medida que JavaScript fue evolucionando, los diseñadores del lenguaje intentaron corregir los fallos de las primeras versiones (anteriores a la ES5). Para mantener la compatibilidad con versiones anteriores, no es posible eliminar las características heredadas, por muy defectuosas que sean. Pero en ES5 y posteriores, los programas pueden optar por el *modo estricto* de JavaScript, en el que se han corregido varios de los primeros errores del lenguaje. El mecanismo para optar es la directiva "use strict" descrita en §5.6.3. En esa sección también se resumen las diferencias entre las versiones anteriores de

-----  
JavaScript y JavaScript estricto. En ES6 y posteriores, el uso de las nuevas características del lenguaje a menudo invoca implícitamente el modo estricto. Por ejemplo, si usas la palabra clave de la clase ES6 o creas un módulo ES6, entonces todo el código dentro de la clase o módulo es automáticamente estricto, y las antiguas características defectuosas no están disponibles en esos contextos. Este libro cubrirá las características heredadas de JavaScript, pero tiene cuidado de señalar que no están disponibles en modo estricto.

lenguaje de programación. Además, hace tiempo que JavaScript ha superado sus raíces de lenguaje de scripting para convertirse en un lenguaje robusto y eficiente de uso general, adecuado para la ingeniería de software seria y los proyectos con bases de código

enormes.

Para ser útil, todo lenguaje debe tener una plataforma, o biblioteca estándar, para realizar cosas como la entrada y salida básicas. El núcleo del lenguaje JavaScript define una API mínima para trabajar con números, texto, matrices, conjuntos, mapas, etc., pero no incluye ninguna funcionalidad de entrada o salida. La entrada y la salida (así como las funciones más sofisticadas, como la creación de

redes, el almacenamiento y los gráficos) son responsabilidad del "entorno anfitrión" en el que se inserta JavaScript.

El entorno anfitrión original para JavaScript fue un navegador web, y éste sigue siendo el entorno de ejecución más común para el código JavaScript. El entorno del navegador web permite que el código JavaScript obtenga entradas del ratón y el teclado del usuario y que realice peticiones HTTP. Y permite que el código JavaScript muestre la salida al usuario con HTML y CSS.

Desde 2010, existe otro entorno de alojamiento para el código JavaScript. En lugar de restringir JavaScript a trabajar con las APIs proporcionadas por un navegador web, Node da acceso a JavaScript a todo el sistema operativo, permitiendo a los programas JavaScript leer y escribir archivos, enviar y recibir datos a través de la red, y hacer y servir peticiones HTTP. Node es una opción popular para implementar servidores web y también una herramienta conveniente para escribir scripts de utilidad simples como una alternativa a los scripts de shell.

La mayor parte de este libro se centra en el propio lenguaje JavaScript. El capítulo [11](#) documenta la biblioteca estándar de JavaScript, [el capítulo 15](#) presenta el entorno del navegador web y [el capítulo 16](#) presenta el entorno de Node.

Este libro cubre primero los fundamentos de bajo nivel, y luego se basa en ellos para llegar a abstracciones más avanzadas y de mayor nivel. Los capítulos están pensados para ser leídos más o menos en orden. Pero el aprendizaje de un nuevo lenguaje de programación nunca es un proceso lineal, y la descripción de un lenguaje tampoco

lo es: cada característica del lenguaje está relacionada con otras características, y este libro está lleno de referencias cruzadas -a veces

hacia atrás y, a veces, hacia adelante, hacia material relacionado. Este capítulo introductorio hace una primera pasada rápida por el lenguaje, introduciendo características clave que facilitarán la comprensión del tratamiento en profundidad en los capítulos siguientes. Si ya es un programador de JavaScript en activo, probablemente pueda saltarse este capítulo. (Aunque podría disfrutar de la lectura [del Ejemplo 1-1](#) al final del capítulo antes de seguir adelante).

## 1.1 Explorando JavaScript

Cuando se aprende un nuevo lenguaje de programación, es importante probar los ejemplos del libro, modificarlos y volver a probarlos para comprobar que se entiende el lenguaje. Para ello, necesitas un intérprete de JavaScript.

La forma más fácil de probar algunas líneas de JavaScript es abrir las herramientas de desarrollo web en tu navegador (con F12, Ctrl-Mayúsculas-I o Comando-Opción-I) y seleccionar la pestaña Consola. A continuación, puedes escribir el código en el indicador y ver los resultados a medida que escribes. Las herramientas de desarrollo del navegador suelen aparecer como paneles en la parte inferior o derecha de la ventana del navegador, pero normalmente puedes separarlas como ventanas independientes (como se muestra en la [Figura 1-1](#)), lo que suele ser muy conveniente.

The screenshot shows the Firefox Developer Tools interface with the 'Console' tab selected. The top bar includes icons for Stop, Reload, and Refresh, followed by the title 'Developer Tools - Node.js - https://nodejs.org/en/'. Below the title is a navigation bar with tabs: Inspector, Console (which is active), Debugger, Style Editor, Performance, Memory, Network, and three more tabs represented by ellipses (...). Underneath the navigation bar is a toolbar with a trash can icon, a 'Filter output' checkbox, and several status indicators: Errors, Warnings, Logs, Info, Debug (which is active), CSS, XHR, Requests, and a 'Persist Logs' checkbox.

```
> let primes = [2, 3, 5, 7];
← undefined
> primes[primes.length-1]
← 7
> primes[0] + primes[1]
← 5
> function fact(x) {
  if (x > 1) return x * fact(x-1);
  else return 1;
}
← undefined
> fact(4)
← 24
> fact(5)
← 120
> fact(6)
← 720
>
```

Figura 1-1. La consola de JavaScript en las herramientas de desarrollo de Firefox

Otra forma de probar el código JavaScript es descargar e instalar Node desde <https://nodejs.org>. Una vez instalado Node en su sistema, sólo tiene que abrir una ventana de Terminal y escribir **node** para iniciar una sesión interactiva de JavaScript como ésta:

```
$ node Bienvenido a Node.js v12.13.0.  
Escriba ".help" para obtener más información.  
> .help .break A veces te quedas atascado, esto te hace salir  
.clear Alias para .break  
.editor Entrar en el modo editor  
.exit Salir de la réplica  
.help Imprimir este mensaje de ayuda  
.load Cargar JS desde un archivo en la sesión REPL  
.save Guarda todos los comandos evaluados en esta sesión REPL en un archivo
```

Presione ^C para abortar la expresión actual, ^D para salir de la repl

```
> dejar x = 2, y = 3;  
undefined > x + y  
5  
> (x === 2) && (y === 3) true  
> (x > 3) || (y < 3) false
```

## 1.2 Hola Mundo

Cuando estés listo para empezar a experimentar con trozos de código más largos, estos entornos interactivos línea por línea pueden dejar de ser adecuados, y probablemente prefieras escribir tu código en un editor de texto. Desde ahí, puedes copiar y pegar en la consola de JavaScript o en una sesión de Node. O puede guardar su código en un archivo (la extensión de nombre de archivo tradicional para el código JavaScript es `.js`) y luego ejecutar ese archivo de código JavaScript con Node:

```
$ node snippet.js
```

Si utilizas Node de una manera no interactiva como ésta, no imprimirá automáticamente el valor de todo el código que ejecutes, así que tendrás que hacerlo tú mismo. Puedes utilizar la función `console.log()` para mostrar el texto y otros valores de JavaScript en tu ventana de terminal o en la consola de las

herramientas de desarrollo de un navegador. Así, por ejemplo, si creas un archivo *hello.js* que contenga esta línea de código

```
console.log("¡Hola Mundo!");
```

y ejecuta el archivo con `node hello.js`, verás el mensaje "¡Hola Mundo!" impreso.

Si quieres ver ese mismo mensaje impreso en la consola JavaScript de un navegador web, crea un nuevo archivo llamado *hola.html*, y pon este texto en él:

```
<script src="hola.js"></script>
```

A continuación, cargue *hello.html* en su navegador web utilizando una URL `file://` como ésta:

```
file:///nombredeusuario/javascript/hola.html
```

Abre la ventana de herramientas de desarrollo para ver el saludo en la consola.

## 1.3 Un recorrido por JavaScript

Esta sección presenta una rápida introducción, mediante ejemplos de código, al lenguaje JavaScript. Después de este capítulo introductorio, nos sumergimos en JavaScript en el nivel más bajo: El [capítulo 2](#) explica cosas como los comentarios de JavaScript, el punto y coma y el conjunto de caracteres Unicode. El [capítulo 3](#) empieza a ser más interesante: explica las variables de JavaScript y los valores que se pueden asignar a esas variables.

A continuación se presenta un código de ejemplo para ilustrar los aspectos más destacados de esos dos capítulos:

```
// Todo lo que sigue a las barras dobles es un comentario en inglés. // Lee atentamente los comentarios: explican el código JavaScript.  
  
// Una variable es un nombre simbólico para un valor.  
// Las variables se declaran con la palabra clave let: let x; // Declara una variable llamada x.  
  
// Se pueden asignar valores a las variables con un signo = x = 0; // Ahora la variable x tiene el valor 0 x => 0: Una variable se evalúa a su valor.  
  
// JavaScript soporta varios tipos de valores x = 1; // Números.  
x = 0.01; // Los números pueden ser enteros o reales.  
x = "hola mundo"; // Cadenas de texto entre comillas.  
x = 'JavaScript'; // Las comillas simples también delimitan cadenas. x = true; // Un valor booleano. x = false; // El otro valor booleano.  
x = null; // Null es un valor especial que significa "sin valor". x = undefined; // Undefined es otro valor especial como null.
```

Otros dos *tipos* muy importantes que los programas de JavaScript pueden manipular son los objetos y las matrices. Estos son los temas de los capítulos [6](#) y [7](#), pero son tan importantes que los verás muchas veces antes de llegar a esos capítulos:

// El tipo de datos más importante de JavaScript es el objeto.  
// Un objeto es una colección de pares nombre/valor, o un mapa de cadena a valor.

```
let book = { // Los objetos están encerrados entre llaves.
    tema: "JavaScript", // La propiedad "topic" tiene valor
    "JavaScript".
    edition: 7 // La propiedad "edition" tiene valor 7 }; // La llave marca el final del
    objeto.
```

// Accede a las propiedades de un objeto con . o []:  
libro.tema // => "JavaScript" libro["edición"] // => 7: otra forma de acceder a los  
valores de las propiedades.

```
libro.autor = "Flanagan"; // Crear nuevas propiedades por asignación.
book.contents = {} // {} es un objeto vacío sin propiedades.
```

// Acceder condicionalmente a las propiedades con ? (ES2020):  
book.contents?.ch01?.sect1 // => undefined: book.contents no tiene la propiedad  
ch01.

// JavaScript también soporta arrays (listas indexadas numéricamente) de valores:

```
let primes = [2, 3, 5, 7]; // Un array de 4 valores, delimitados con [ y ].
primos[0] // => 2: el primer elemento (índice 0) de la matriz.
primos.longitud // => 4: cuántos elementos hay en el array.
primes[primes.length-1] // => 7: el último elemento de la matriz.
primos[4] = 9; // Añadir un nuevo elemento por asignación.
primos[4] = 11; // O alterar un elemento existente por asignación.
let empty = []; // [] es un array vacío sin elementos. empty.length // => 0 // Los
arrays y objetos pueden contener otros arrays y objetos:
```

```
let points = [ // Un array con 2 elementos. {x: 0, y: 0}, // Cada elemento es un objeto.  
{x: 1, y: 1} ]; let data = { // Un objeto con 2 propiedades trial1: [[1,2], [3,4]], // El valor de cada propiedad es un array.  
trial2: [[2,3], [4,5]] // Los elementos de las matrices son matrices.  
};
```

## SINTAXIS DE LOS COMENTARIOS EN LOS

Usted puede haber notado en el código anterior que algunos de los comentarios comienzan con una flecha (`// =>`). Estos muestran el valor producido por el código antes del comentario y son mi intento de emular un entorno interactivo de JavaScript como una consola de navegador web en un libro impreso.

Esos `// =>` comentarios también sirven para *afirmación*, he escrito una herramienta que prueba el código y verifica que produce el valor especificado en el comentario. Esto debería ayudar, espero, a reducir los errores en el libro.

Hay dos estilos relacionados de comentario/afirmación. Si ve un comentario de la forma `a == 42`, significa que después de que se ejecute el código antes del comentario, `a` es igual a 42. Si ve un comentario de la forma `/! a == 42`, significa que el código de la línea anterior al comentario lanza un excepción (y el resto del comentario después del signo de exclamación suele explicar qué tipo de se lanza la excepción).

Verás que estos comentarios se utilizan a lo largo del libro.

La sintaxis ilustrada aquí para enumerar los elementos del array dentro de las llaves cuadradas o para asignar los nombres de las propiedades de los objetos a los valores de las propiedades dentro de las llaves rizadas se conoce como expresión *inicializadora*, y es sólo uno de los temas del [capítulo 4](#). Una *expresión* es una frase de JavaScript que puede ser *evaluada* para producir un valor. Por ejemplo, el uso de `.y` y `[]` para referirse al valor de una propiedad de objeto o de un elemento de matriz es una expresión.

Una de las formas más comunes de formar expresiones en JavaScript es utilizar *operadores*:

```

// Los operadores actúan sobre los valores (los operandos) para producir un nuevo valor.

// Los operadores aritméticos son de los más sencillos:
3 + 2 //=> 5: adición
3 - 2 //=> 1: resta
3 * 2 //=> 6: multiplicación
3 / 2 //=> 1.5: división puntos[1].x - puntos[0].x // => 1: operandos más complicados también funcionan
"3" + "2" //=> "32": + suma números, concatena cadenas

// JavaScript define algunos operadores aritméticos abreviados let count = 0; //
Define una variable count++; // Incrementa la variable count--; // Disminuye la variable count += 2; // Suma 2: igual que count = count + 2; count *= 3; // Multiplica por 3: igual que count = count * 3; count //=> 6: los nombres de las variables también son expresiones.

// Los operadores de igualdad y relacionales prueban si dos valores son iguales, //
desiguales, menores que, mayores que, etc. Se evalúan como verdadero o falso. let x = 2, y = 3; // Estos signos = son pruebas de asignación, no de igualdad x === y // => falso: igualdad x !== y // => verdadero: desigualdad x < y // => verdadero: menor que x <= y // => verdadero: menor o igual que x > y // => falso: mayor que x >= y // => falso: mayor o igual que "dos" === "tres" // => falso: las dos cadenas son diferentes "dos" > "tres" // => verdadero: "tw" es alfabéticamente mayor que "th" false === (x > y) // => verdadero: false es igual a false

// Los operadores lógicos combinan o invierten los valores booleanos
(x === 2) && (y === 3) // => verdadero: ambas comparaciones son verdaderas. && es AND (x > 3) || (y < 3) // => falso: ninguna de las comparaciones es verdadera. || es OR ! (x === y) // => verdadero: ! invierte un valor booleano

```

Si las expresiones de JavaScript son como frases, *las sentencias* de JavaScript son como oraciones completas. Las sentencias son el tema del [capítulo 5](#). A grandes rasgos, una expresión es algo que calcula un valor pero no *hace* nada: no altera el estado del programa de ninguna manera. Las declaraciones, por otro lado, no tienen un valor, pero sí alteran el estado. Ya has visto las declaraciones de variables y las sentencias de asignación. La otra gran categoría de sentencias son las *estructuras de control*, como los condicionales y los bucles. Verás ejemplos más adelante, después de que cubramos las funciones.

Una función es un bloque de código JavaScript con nombre y parámetros que se define una vez y que se puede invocar una y otra vez. Las funciones no se tratan formalmente hasta [el capítulo 8](#), pero al igual que los objetos y las matrices, las verás muchas veces antes de llegar a ese capítulo. He aquí algunos ejemplos sencillos:

```
// Las funciones son bloques de código JavaScript parametrizados que podemos invocar. function plus1(x) { // Definir una función llamada "plus1" con el parámetro "x" return x + 1; // Devolver un valor uno mayor que el valor pasado } // Las funciones se encierran entre llaves

plus1(y) // => 4: y es 3, por lo que esta invocación devuelve 3+1

let square = function(x) { // Las funciones son valores y pueden asignarse a vars
    return x * x; // Calcula el valor de la función
}; // El punto y coma marca el final de la asignación.

cuadrado(más1(y)) // => 16: invocar dos funciones en una sola expresión
```

En ES6 y posteriores, existe una sintaxis abreviada para definir funciones. Esta sintaxis concisa utiliza `=>` para separar la lista de argumentos del cuerpo de la función, por lo que las funciones definidas de este modo se conocen como funciones flecha. Las funciones en flecha se utilizan normalmente cuando se quiere pasar una función sin nombre como argumento a otra función. El código anterior se ve así cuando se reescribe para usar funciones de flecha:

```
const plus1 = x => x + 1; // La entrada x se asigna a la salida x + 1 const square = x => x
* x; // La entrada x se asigna a la salida x * x plus1(y) // => 4: la invocación de la función es la misma

cuadrado(más1(y)) // => 16
```

Cuando usamos funciones con objetos, obtenemos *métodos*:

```
// Cuando se asignan funciones a las propiedades de un objeto, las llamamos //  
"métodos". Todos los objetos de JavaScript (incluidos los arrays) tienen métodos:  
let a = []; // Crear una matriz vacía  
a. push(1,2,3); // El método push() añade elementos a un array  
a. reverse(); // Otro método: invertir el orden de los elementos  
  
// También podemos definir nuestros propios métodos. La palabra clave "this" se  
refiere al objeto // sobre el que se define el método: en este caso, el array de puntos  
de antes.  
puntos. dist = function() { // Definir un método para calcular
```

```
distancia entre puntos let p1 = this[0]; // Primer elemento del array sobre el que se  
invoca let p2 = this[1]; // Segundo elemento del objeto "this" let a = p2.x-p1.x; //  
Diferencia en coordenadas x let b = p2.y-p1.y; // Diferencia en coordenadas y return  
Math. sqrt(a*a + // El teorema de Pitágoras b*b); // Math.sqrt() calcula la raíz cuadrada  
}; points. dist() //=> Math.sqrt(2): distancia entre nuestros 2 puntos
```

Ahora, como se prometió, aquí hay algunas funciones cuyos cuerpos demuestran declaraciones comunes de la estructura de control de JavaScript:

// Las sentencias de JavaScript incluyen condicionales y bucles utilizando la sintaxis // de C, C++, Java y otros lenguajes.

```
function abs(x) { // Una función para calcular el valor
    // absoluto.
    if (x >= 0) { // La sentencia if...
        return x; // ejecuta este código si la comparación es verdadera.
    }
    // Este es el final de la cláusula if.
    else { // La cláusula opcional else ejecuta su código si return -x; // la
        // comparación es falsa.
    }
    // Las llaves son opcionales cuando hay una declaración por cláusula.
}
// Observe las declaraciones de retorno anidadas dentro de if/else.
abs(-10) === abs(10) // => true
```

**function** sum(array) { // Calcula la suma de los elementos de un array let sum = 0; // Empieza con una suma inicial de 0. **for(let x of array)** { // Repasa el array, asignando cada elemento a x.

```

        sum += x; // Añadir el valor del elemento a la suma. }           // Este es el
final del bucle. return sum; // Devuelve la suma. } sum(primos) // => 28: suma de los
5 primeros primos 2+3+5+7+11

function factorial(n) { // Una función para calcular factoriales let product = 1; //
Empieza con un producto de 1 while(n > 1) { // Repite las declaraciones en {} while expr
in () is true product *= n; // Atajo para product = product * n; n--; // Atajo para n = n -
1 }           // Fin del bucle return product; // Devuelve el producto } factorial(4) //
=> 24: 1*4*3*2

function factorial2(n) { // Otra versión usando un bucle diferente let i,
product = 1; // Empieza con 1
    for(i=2; i <= n; i++) // Incrementa automáticamente i desde 2 hasta n producto *= i;
// Hace esto cada vez. {} no es necesario para bucles de 1 línea return product; //
Devuelve el factorial } factorial2(5) // => 120: 1*2*3*4*5

```

JavaScript soporta un estilo de programación orientado a objetos, pero es significativamente diferente de los lenguajes de programación orientados a objetos "clásicos". [El capítulo 9](#) cubre la programación orientada a objetos en JavaScript en detalle, con muchos ejemplos. Aquí hay un ejemplo muy simple que demuestra cómo definir una clase de JavaScript para representar puntos geométricos 2D. Los objetos que son instancias de esta clase tienen un único método, llamado `distance()`, que calcula la distancia del punto desde el origen:

```

class Point { // Por convención, los nombres de las clases van en mayúsculas.
    constructor(x, y) { // Función constructora para inicializar nuevas instancias.
        this.x = x; // Esta palabra clave es el nuevo objeto que se inicializa.
        this.y = y; // Almacena los argumentos de la función como propiedades del
        objeto. } // No es necesario el retorno en las funciones
        constructoras.

    distancia() { // Método para calcular la distancia del origen al punto.
        return Math.sqrt( // Devuelve la raíz cuadrada de  $x^2 + y^2$ .
            this.x * this.x + // esto se refiere al objeto Punto sobre el que se invoca este.
            y * this.y // el método de la distancia. );
    }

    // Utiliza la función constructora Point() con "new" para crear
    // Objetos punto let p = new Point(1, 1); // El punto geométrico (1,1).

    // Ahora utiliza un método del objeto Punto p
    p.distancia() // => Math.SQRT2
}

```

Este recorrido introductorio por la sintaxis y las capacidades fundamentales de JavaScript termina aquí, pero el libro continúa con capítulos autocontenido que cubren características adicionales del lenguaje:

## Capítulo 10, Módulos

Muestra cómo el código JavaScript de un archivo o script puede utilizar funciones y clases JavaScript definidas en otros archivos o scripts.

## Capítulo 11, La biblioteca estándar de JavaScript

Cubre las funciones y clases incorporadas que están disponibles para todos los programas de JavaScript. Esto incluye importantes estructuras de datos como mapas y conjuntos, una clase de expresión regular para la coincidencia de patrones textuales, funciones para serializar estructuras de datos de JavaScript y mucho más.

### *Capítulo 12, Iteradores y generadores*

Explica cómo funciona el bucle for/of y cómo puedes hacer tus propias clases iterables con for/of. También cubre las funciones generadoras y la sentencia yield.

### *Capítulo 13, JavaScript asíncrono*

Este capítulo es una exploración en profundidad de la programación asíncrona en JavaScript, cubriendo las devoluciones de llamada y los eventos, las APIs basadas en promesas, y las palabras clave async y await. Aunque el núcleo del lenguaje JavaScript no es asíncrono, las APIs asíncronas son las predeterminadas tanto en los navegadores web como en Node, y este capítulo explica las técnicas para trabajar con esas APIs.

### *Capítulo 14, Metaprogramación*

Introduce una serie de características avanzadas de JavaScript que pueden ser de interés para los programadores que escriben bibliotecas de código para que otros programadores de JavaScript las utilicen.

### *Capítulo 15, JavaScript en los navegadores web*

Introduce el entorno del navegador web, explica cómo los navegadores web ejecutan el código JavaScript y cubre las más importantes de las muchas APIs definidas por los navegadores web. Este es, con mucho, el capítulo más largo del libro.

### *Capítulo 16, JavaScript del lado del servidor con Node*

Introduce el entorno de host Node, cubriendo el modelo de programación fundamental y las estructuras de datos y APIs que son más importantes de entender.

### Capítulo 17, Herramientas y extensiones de JavaScript

Cubre herramientas y extensiones del lenguaje que vale la pena conocer porque son ampliamente utilizadas y pueden hacer de usted un programador más productivo.

## 1.4 Ejemplo: Histogramas de frecuencia de caracteres

Este capítulo concluye con un programa JavaScript corto pero no trivial. [El ejemplo 1-1](#) es un programa Node que lee texto de la entrada estándar, calcula un histograma de frecuencia de caracteres a partir de ese texto y luego imprime el histograma. Podría invocar el programa así para analizar la frecuencia de caracteres de su propio código fuente:

```
$ node charfreq.js < charfreq.js
T: ##### 11.22%
E: ##### 10.15%
R: ##### 6.68%
S: ##### 6.44%
A: ##### 6.16%
N: ##### 5.81%
O: ##### 5.45%
I: ##### 4.54%
H: #### 4.07%
C: ### 3.36%
L: ### 3.20%
U: ### 3.08%
/: ## 2.88%
```

Este ejemplo utiliza una serie de características avanzadas de JavaScript y pretende demostrar cómo pueden ser los programas de JavaScript del mundo real. No debes esperar entender todo el código

todavía, pero ten por seguro que todo se explicará en los capítulos siguientes.

### *Ejemplo 1-1. Cálculo de histogramas de frecuencia de caracteres con JavaScript*

```
/**  
 * Este programa Node lee el texto de la entrada estándar, calcula la frecuencia  
 * de cada letra de ese texto, y muestra un histograma de los caracteres más *  
frecuentes. Requiere el Nodo 12 o superior para funcionar. *  
 * En un entorno de tipo Unix se puede invocar el programa así:  
 * node charfreq.js < corpus.txt  
 */  
  
// Esta clase extiende Map para que el método get() devuelva el valor // especificado en  
lugar de null cuando la clave no está en el mapa class DefaultMap extends Map {  
constructor(defaultValue) { super(); // Invoca el constructor de la superclase this.  
defaultValue = defaultValue; // Recuerda el valor por defecto }  
  
get(key) { if (this. has(key)) { // Si la clave ya está en el mapa return super. get(key);  
// devuelve su valor desde la superclase. } else { return this. defaultValue; // En  
caso contrario devuelve el valor por defecto  
}  
}
```

```
        }
    }

// Esta clase calcula y muestra los histogramas de frecuencia de letras class Histogram {
constructor() { this.letterCounts = new DefaultMap(0); // Mapa de letras a recuentos this.
totalLetters = 0; // Cuántas letras en total }

// Esta función actualiza el histograma con las letras del texto.
add(text) {
    // Eliminar los espacios en blanco del texto y convertirlo en mayúsculas text =
text.replace(/\s/g, "").toUpperCase();

    // Ahora recorre los caracteres del texto for(let character of text) {
        let count = this.letterCounts.get(character); // Obtener la cuenta antigua
this.letterCounts.set(character, count+1); // Incrementarlo this.
totalLetters++;
    }

    // Convertir el histograma en una cadena que muestre un gráfico ASCII toString() { //
Convertir el Mapa en una matriz de matrices [clave,valor] let entries = [... this.
letterCounts];

    // Ordena el array por número, y luego por entradas alfabéticas. sort((a,b) => { // Una
función para definir el orden de clasificación.
        if (a[1] === b[1]) { // Si las cuentas son iguales return a[0] < b[0] ? -1 : 1; // ordenar
alfabéticamente.
        } else { // Si las cuentas
```

```

differ return b[1] - a[1]; // ordenar por el mayor número.      }
});

// Convertir los recuentos en porcentajes for(let entry of entries) { entry[1] =
entry[1] / this.totalLetters*100; }

// Elimina las entradas inferiores al 1% entries = entries.filter(entry =>
entry[1] >= 1);

// Ahora convierta cada entrada en una línea de texto let lines = entries.map(
([l,n]) => `${l}: ${"#".repeat(Math.round(n))}`
${n.toFixed(2)}%
);

// Y devuelve las líneas concatenadas, separadas por caracteres de nueva línea.
return lines.join("\n");
}

// Esta función asíncrona (con devolución de promesa) crea un objeto Histograma,
// lee asíncronamente trozos de texto de la entrada estándar, y añade esos trozos a
// el histograma. Cuando llega al final del flujo, devuelve este histograma async function
histogramFromStdin() { process.stdin.setEncoding("utf-8"); // Lee cadenas Unicode, no
bytes let histogram = new Histogram(); for await (let chunk of process.stdin) { histogram.
add(chunk); } return histogram; } // Esta última línea de código es el cuerpo principal del
programa.

// Crea un objeto Histograma a partir de la entrada estándar, y luego imprime el
histograma.
histogramFromStdin().then(histogram => { console.log(histogram.toString()); });

```

## 1.5 Resumen

Este libro explica JavaScript de abajo a arriba. Esto significa que empezamos con detalles de bajo nivel como comentarios, identificadores, variables y tipos; luego pasamos a expresiones, declaraciones, objetos y funciones; y después cubrimos las abstracciones de alto nivel del lenguaje como las clases y los módulos. Me tomo muy en serio la palabra "*definitivo*" que aparece en el título de este libro, y los próximos capítulos explican el lenguaje con un nivel de detalle que puede resultar desconcertante al principio. Sin embargo, el verdadero dominio de JavaScript requiere una comprensión de los detalles, y espero que se tome el tiempo de leer este libro de principio a fin. Pero, por favor, no sientas que tienes que hacerlo en tu primera lectura. Si se siente atascado en una sección, simplemente pase a la siguiente. Podrá volver y dominar los detalles una vez que tenga un conocimiento práctico de la lengua en su conjunto.

## Capítulo 2. Estructura léxica

---

La estructura léxica de un lenguaje de programación es el conjunto de reglas elementales que especifican cómo se escriben los programas en ese lenguaje. Es la sintaxis de más bajo nivel de un lenguaje: especifica el aspecto de los nombres de las variables, los caracteres delimitadores para los comentarios y cómo se separa una sentencia del programa de la siguiente, por ejemplo. Este breve capítulo documenta la estructura léxica de JavaScript.

Abarca:

- Distinción entre mayúsculas y minúsculas, espacios y saltos de línea
- Comentarios
- Literales
- Identificadores y palabras reservadas
- Unicode

Punto y coma opcional

## 2.1 El texto de un programa JavaScript

JavaScript es un lenguaje que distingue entre mayúsculas y minúsculas. Esto significa que las palabras clave del lenguaje, las variables, los nombres de las funciones y otros *identificadores* deben escribirse siempre con una mayúscula consistente. La palabra clave `while`, por ejemplo, debe escribirse "while", no "While" o "MIENTRAS". Del mismo modo, `online`, `Online`, `OnLine` y `ONLINE` son cuatro nombres de variable distintos.

JavaScript ignora los espacios que aparecen entre los tokens en los programas. En su mayor parte, JavaScript también ignora los saltos de línea (pero véase §2.6 para una excepción). Dado que puede utilizar espacios y saltos de línea libremente en sus programas, puede formatear y aplicar sangrías a sus programas de forma ordenada y coherente para que el código sea fácil de leer y comprender.

Además del carácter de espacio normal (\u0020), JavaScript también reconoce los tabuladores, diversos caracteres de control ASCII y varios caracteres de espacio Unicode como espacios en blanco. JavaScript reconoce las nuevas líneas, los retornos de carro

y una secuencia de retorno de carro/salto de línea como terminadores de línea.

## 2.2 Observaciones

JavaScript admite dos estilos de comentarios. Cualquier texto entre un // y el final de una línea se trata como un comentario y es ignorado por JavaScript. Cualquier texto entre los caracteres /\* y \*/ también se trata como un comentario; estos comentarios pueden abarcar varias líneas pero no pueden estar anidados. Las siguientes líneas de código son comentarios legales de JavaScript:

```
// Este es un comentario de una sola línea.  
  
/* Esto también es un comentario */ // y aquí hay otro comentario.  
  
/*  
 * Esto es un comentario de varias líneas. Los caracteres * adicionales al principio de *  
 cada línea no son una parte necesaria de la sintaxis; ¡sólo se ven bien! */
```

## 2.3 Literales

Un *literal* es un valor de datos que aparece directamente en un programa. Los siguientes son todos literales:

```
12 // El número doce  
1.2 // El número uno punto dos "hola mundo" // Una cadena de texto  
'Hola' // Otra cadena true // Un valor booleano false // El otro valor  
booleano null // Ausencia de un objeto
```

Los detalles completos sobre los literales numéricos y de cadena aparecen en [el capítulo 3](#).

## 2.4 Identificadores y palabras reservadas

Un *identificador* es simplemente un nombre. En JavaScript, los identificadores se utilizan para nombrar constantes, variables, propiedades, funciones y clases y para proporcionar etiquetas para ciertos bucles en el código JavaScript. Un identificador de JavaScript debe comenzar con una letra, un guión bajo (\_) o un signo de dólar (\$). Los caracteres siguientes pueden ser letras, dígitos, guiones bajos o signos de dólar. (Los dígitos no están permitidos como primer carácter para que JavaScript pueda distinguir fácilmente los identificadores de los números). Todos estos son identificadores legales:

```
i mi_nombre_de_la_variable v13 _mentira  
$str
```

Como cualquier lenguaje, JavaScript reserva ciertos identificadores para su uso por el propio lenguaje. Estas "palabras reservadas" no pueden utilizarse como identificadores normales. Se enumeran en la siguiente sección.

### 2.4.1 Palabras reservadas

Las siguientes palabras forman parte del lenguaje JavaScript. Muchas de ellas (como if, while y for) son palabras clave reservadas que no deben usarse como nombres de constantes, variables, funciones o clases (aunque todas pueden usarse como nombres de propiedades dentro de un objeto). Otras (como from, of, get y set) se utilizan en contextos limitados sin ambigüedad sintáctica y son perfectamente legales como identificadores. Otras palabras clave (como let) no pueden reservarse completamente para mantener la compatibilidad con programas anteriores, por lo que existen

complejas reglas que rigen cuándo pueden utilizarse como identificadores y cuándo no. (let puede usarse como nombre de variable si se declara con var fuera de una clase, por ejemplo, pero no si se declara dentro de una clase o con const). Lo más sencillo es evitar el uso de cualquiera de estas palabras como identificadores, excepto from, set y target, que son seguros de usar y ya son de uso común.

```
as const export get null target void await continue extends if of this while await debugger false import return throw with break default finally in set true yield case delete for instanceof static try catch do from let super typeof class else function new switch var
```

JavaScript también se reserva o restringe el uso de ciertas palabras clave que no se utilizan actualmente en el lenguaje pero que podrían utilizarse en futuras versiones:

```
enum implements interface package private protected public
```

Por razones históricas, los argumentos y eval no están permitidos como identificadores en ciertas circunstancias y es mejor evitarlos por completo.

## 2.5 Unicode

Los programas de JavaScript se escriben utilizando el conjunto de caracteres Unicode, y se puede utilizar cualquier carácter Unicode en las cadenas y los comentarios. Por razones de portabilidad y facilidad de edición, es habitual utilizar sólo letras y dígitos ASCII en los identificadores. Pero esto es sólo una convención de programación, y el lenguaje permite letras, dígitos e ideogramas Unicode (pero no emojis) en los identificadores. Esto significa que los programadores

pueden utilizar símbolos matemáticos y palabras de idiomas no ingleses como constantes y variables:

```
const π = 3,14; const sí =
true;
```

## 2.5.1 Secuencias de escape Unicode

Algunos equipos y programas informáticos no pueden mostrar, introducir o procesar correctamente el conjunto completo de caracteres Unicode. Para ayudar a los programadores y a los sistemas que utilizan tecnología antigua, JavaScript define secuencias de escape que permiten escribir caracteres Unicode utilizando sólo caracteres ASCII. Estos escapes Unicode comienzan con los caracteres \u y van seguidos de exactamente cuatro dígitos hexadecimales (utilizando letras mayúsculas o minúsculas A-F) o de uno a seis dígitos hexadecimales encerrados entre llaves. Estos escapes Unicode pueden aparecer en

Literales de cadena de JavaScript, literales de expresiones regulares e identificadores (pero no en palabras clave del lenguaje). El escape Unicode para el carácter "é", por ejemplo, es \u00E9; aquí hay tres formas diferentes de escribir un nombre de variable que incluya este carácter:

```
let café = 1; // Definir una variable usando un carácter Unicode caf\u00e9 // => 1;
acceder a la variable usando una secuencia de escape caf\u{E9} // => 1; otra forma
de la misma secuencia de escape
```

Las primeras versiones de JavaScript sólo admitían la secuencia de escape de cuatro dígitos. La versión con llaves se introdujo en ES6 para soportar mejor los puntos de código Unicode que requieren más de 16 bits, como los emoji:

```
console.log("\u{1F600}"); // Imprime un emoji de cara soniente
```

Los escapes Unicode también pueden aparecer en los comentarios, pero como los comentarios se ignoran, simplemente se tratan como caracteres ASCII en ese contexto y no se interpretan como Unicode.

## 2.5.2 Normalización Unicode

Si utiliza caracteres no ASCII en sus programas de JavaScript, debe saber que Unicode permite más de una forma de codificar el mismo carácter. La cadena "é", por ejemplo, puede codificarse como el único carácter Unicode \u00E9 o como una "e" ASCII normal seguida de la marca de combinación del acento agudo \u0301. Estas dos codificaciones suelen tener el mismo aspecto cuando se muestran en un editor de texto, pero tienen codificaciones binarias diferentes, lo que significa que son consideradas diferentes por JavaScript, lo que puede dar lugar a programas muy confusos:

```
const café = 1; // Esta constante se llama "caf\u{e9}" const café = 2; // Esta constante es diferente: "cafe\u{301}" café // => 1: esta constante tiene un valor café // => 2: esta constante indistinta tiene un valor diferente
```

El estándar Unicode define la codificación preferida para todos los caracteres y especifica un procedimiento de normalización para convertir el texto en una forma canónica adecuada para las comparaciones. JavaScript asume que el código fuente que interpreta ya ha sido normalizado y *no* realiza ninguna normalización por sí mismo. Si tiene previsto utilizar caracteres Unicode en sus programas de JavaScript, debe asegurarse de que su editor o alguna otra herramienta realice la normalización Unicode de su código fuente para evitar que acabe con identificadores diferentes pero visualmente indistinguibles.

## 2.6 Punto y coma opcional

Como muchos lenguajes de programación, JavaScript utiliza el punto y coma (;) para separar las sentencias (véase [el capítulo 5](#)) entre sí.

Esto es importante para dejar claro el significado de su código: sin un separador, el final de una sentencia podría parecer el principio de la siguiente, o viceversa. En JavaScript, normalmente se puede omitir el punto y coma entre dos sentencias si éstas están escritas en líneas separadas. (También se puede omitir el punto y coma al final de un programa o si el siguiente elemento del programa es una llave de cierre: }).) Muchos

Los programadores de JavaScript (y el código de este libro) utilizan el punto y coma para marcar explícitamente el final de las sentencias, incluso cuando no es necesario. Otro estilo es omitir el punto y coma siempre que sea posible, utilizándolo sólo en las pocas situaciones que lo requieren. Sea cual sea el estilo que elija, hay algunos detalles que debe entender sobre los puntos y comas opcionales en JavaScript.

Considere el siguiente código. Dado que las dos sentencias aparecen en líneas separadas, se puede omitir el primer punto y coma:

```
a = 3; b  
= 4;
```

Sin embargo, si se escribe así, es necesario el primer punto y coma:

```
a = 3; b = 4;
```

Tenga en cuenta que JavaScript no trata cada salto de línea como un punto y coma: normalmente trata los saltos de línea como

puntos y comas sólo si no puede analizar el código sin añadir un punto y coma implícito. Más formalmente (y con tres excepciones que se describen más adelante), JavaScript trata un salto de línea como un punto y coma si el siguiente carácter no espacial no puede interpretarse como una continuación de la sentencia actual. Considere el siguiente código:

```
let a a = 3  
console.log(a)
```

JavaScript interpreta este código así:

```
let a; a = 3; console.log(a);
```

JavaScript trata el primer salto de línea como un punto y coma porque no puede analizar el código que deja a a sin punto y coma. La segunda a podría estar sola como la declaración a;, pero JavaScript no trata el segundo salto de línea como un punto y coma porque puede continuar analizando la declaración más larga a = 3;.

Estas reglas de terminación de sentencias conducen a algunos casos sorprendentes. Este código parece dos sentencias separadas por una nueva línea:

```
let y = x + f (a+b). toString()
```

Pero los paréntesis de la segunda línea de código pueden interpretarse como una invocación a la función f de la primera línea, y JavaScript interpreta el código así:

```
let y = x + f(a+b). toString();
```

Lo más probable es que esta no sea la interpretación pretendida por el autor del código. Para que funcione como dos sentencias separadas, en este caso se requiere un punto y coma explícito.

En general, si una sentencia comienza con (, [, + o -, existe la posibilidad de que se interprete como una continuación de la sentencia anterior. Las sentencias que comienzan con /, + y - son bastante raras en la práctica, pero las que comienzan con ( y [ no son en absoluto infrecuentes, al menos en algunos estilos de programación en JavaScript. A algunos programadores les gusta poner un punto y coma defensivo al principio de cualquier sentencia de este tipo para que siga funcionando correctamente aunque se modifique la sentencia que la precede y se elimine el punto y coma anterior:

```
let x = 0 // Se omite el punto y coma aquí ;[x,x+1,x+2].forEach(console.log) //  
Defensivo ; mantiene esta declaración separada
```

Hay tres excepciones a la regla general de que JavaScript interpreta los saltos de línea como punto y coma cuando no puede analizar la segunda línea como una continuación de la sentencia de la primera línea. La primera excepción afecta a las sentencias return, throw, yield, break y continue

(véase [el capítulo 5](#)). Estas sentencias suelen ir solas, pero a veces van seguidas de un identificador o una expresión. Si aparece un salto de línea después de cualquiera de estas palabras (antes de cualquier otro token), JavaScript siempre interpretará ese salto de línea como un punto y coma. Por ejemplo, si se escribe

```
devolverá el  
valor de  
verdad;
```

JavaScript supone que querías decir:

**volver; verdadero;**

Sin embargo, probablemente querías decir:

**devolverá el valor de verdad;**

Esto significa que no debe insertar un salto de línea entre return, break o continue y la expresión que sigue a la palabra clave. Si insertas un salto de línea, es probable que tu código falle de una manera no obvia y difícil de depurar.

La segunda excepción son los operadores ++ y -- ([§4.8](#)). Estos operadores pueden ser operadores prefijos que aparecen antes de una expresión u operadores postfijos que aparecen después de una expresión. Si desea utilizar cualquiera de estos operadores como operadores postfijos, deben aparecer en la misma línea que la expresión a la que se aplican. La tercera excepción se refiere a las funciones definidas mediante la sintaxis concisa "flecha": la propia flecha => debe aparecer en la misma línea que la lista de parámetros.

## 2.7 Resumen

Este capítulo ha mostrado cómo se escriben los programas de JavaScript en el nivel más bajo. El siguiente capítulo nos lleva un paso más arriba y presenta los tipos y valores primitivos (números, cadenas, etc.) que sirven como unidades básicas de cálculo para los programas de JavaScript.



# Capítulo 3. Tipos, valores y variables

---

Los programas de ordenador funcionan manipulando valores, como el número 3,14 o el texto "Hola Mundo". Las clases de valores que pueden representarse y manipularse en un lenguaje de programación se conocen como tipos, y una de las características más fundamentales de un lenguaje de programación es el conjunto de tipos que admite. Cuando un programa necesita retener un valor para utilizarlo en el futuro, asigna el valor a (o "almacena" el valor en) una variable. Las variables tienen nombres, y permiten utilizar esos nombres en nuestros programas para referirse a los valores. El funcionamiento de las variables es otra característica fundamental de cualquier lenguaje de programación. Este capítulo explica los tipos, valores y variables en JavaScript. Comienza con una visión general y algunas definiciones.

## 3.1 Visión general y definiciones

Los tipos de JavaScript pueden dividirse en dos categorías: *tipos primitivos* y *tipos de objeto*. Los tipos primitivos de JavaScript incluyen números, cadenas de texto (conocidas como strings) y valores de verdad booleanos (conocidos como booleans). Una parte importante de este capítulo está dedicada a una explicación detallada de los tipos numéricos ([§3.2](#)) y de cadena ([§3.3](#)) en JavaScript.

Los booleanos se tratan en [§3.4](#).

Los valores especiales de JavaScript null y undefined son valores primitivos, pero no son números, cadenas o booleanos. Cada valor se considera normalmente como el único miembro de su propio tipo especial. En [§3.5](#) hay más información sobre null y undefined. ES6 añade un nuevo tipo de propósito especial, conocido como Symbol, que permite la definición de extensiones del lenguaje sin dañar la compatibilidad hacia atrás. Los símbolos se tratan brevemente en [§3.6](#).

Cualquier valor de JavaScript que no sea un número, una cadena, un booleano, un símbolo, null o undefined es un objeto. Un objeto (es decir, un miembro del tipo *object*) es una colección de *propiedades* donde cada propiedad tiene un nombre y un valor (ya sea un valor primitivo u otro objeto). Un objeto muy especial, el *objeto global*, se trata en [§3.7](#), pero una cobertura más general y detallada de los objetos se encuentra en [el Capítulo 6](#).

Un objeto ordinario de JavaScript es una colección desordenada de valores con nombre. El lenguaje también define un tipo especial de objeto, conocido como array, que representa una colección ordenada de valores numerados. El lenguaje JavaScript incluye una sintaxis especial para trabajar con arrays, y los arrays tienen un comportamiento especial que los distingue de los objetos ordinarios. Los arrays son el tema del [capítulo 7](#).

Además de los objetos básicos y las matrices, JavaScript define otros tipos de objetos útiles. Un objeto Set representa un conjunto de valores. Un objeto Map representa un mapeo de claves a valores. Varios tipos de "matrices tipificadas" facilitan las operaciones con

matrices de bytes y otros datos binarios. El tipo RegExp representa patrones textuales y permite realizar sofisticadas operaciones de coincidencia, búsqueda y sustitución de cadenas. El tipo Date representa fechas y horas y admite la aritmética de fechas rudimentaria. El tipo Error y sus subtipos representan errores que pueden surgir al ejecutar código JavaScript. Todos estos tipos se tratan en [el capítulo 11](#).

JavaScript se diferencia de otros lenguajes más estáticos en que las funciones y las clases no son sólo parte de la sintaxis del lenguaje: son en sí mismas valores que pueden ser manipulados por los programas de JavaScript. Como cualquier valor de JavaScript que no sea un valor primitivo, las funciones y las clases son un tipo de objeto especializado. Se tratan en detalle en los capítulos [8](#) y [9](#).

El intérprete de JavaScript realiza la recolección automática de basura para la gestión de la memoria. Esto significa que un programador de JavaScript generalmente no necesita preocuparse por la destrucción o desasignación de objetos u otros valores. Cuando un valor ya no es accesible -cuando un programa ya no tiene forma de referirse a él- el intérprete sabe que no puede volver a utilizarse y recupera automáticamente la memoria que estaba ocupando. (Los programadores de JavaScript a veces tienen que tener cuidado para asegurarse de que los valores no permanezcan inadvertidamente accesibles -y, por tanto, no recuperables- durante más tiempo del necesario).

JavaScript soporta un estilo de programación orientado a objetos. En términos generales, esto significa que en lugar de tener funciones definidas globalmente para operar con valores de varios tipos, los

propios tipos definen métodos para trabajar con los valores. Para ordenar los elementos de un array `a`, por ejemplo, no pasamos `a` a una función `sort()`. En su lugar, invocamos el método `sort()` de `a`:

`a.sort(); // La versión orientada a objetos de sort(a).`

La definición de métodos se trata en [el capítulo 9](#). Técnicamente, sólo los objetos de JavaScript tienen métodos. Pero los números, las cadenas, los valores booleanos y los símbolos se comportan como si tuvieran métodos. En JavaScript, `null` y `undefined` son los únicos valores sobre los que no se pueden invocar métodos.

Los tipos de objeto de JavaScript son *mutables* y sus tipos primitivos son *inmutables*. Un valor de un tipo mutable puede cambiar: un programa de JavaScript puede cambiar los valores de las propiedades de los objetos y los elementos de las matrices. Los números, los booleanos, los símbolos, los nulos y los indefinidos son inmutables; ni siquiera tiene sentido hablar de cambiar el valor de un número, por ejemplo. Las cadenas pueden considerarse como matrices de caracteres, y es de esperar que sean mutables. En JavaScript, sin embargo, las cadenas son inmutables: se puede acceder al texto en cualquier índice de una cadena, pero JavaScript no proporciona ninguna forma de alterar el texto de una cadena existente. Las diferencias entre los valores mutables e inmutables se analizan con más detalle en [el apartado 3.8](#).

JavaScript convierte libremente los valores de un tipo a otro. Si un programa espera una cadena, por ejemplo, y usted le da un número, automáticamente convertirá el número en una cadena. Y si utiliza un valor no booleano donde se espera un booleano, JavaScript lo convertirá en consecuencia. Las reglas de conversión de valores se explican en [§3.9](#). Las reglas liberales de conversión de valores de

JavaScript afectan a su definición de igualdad, y el operador de igualdad `==` realiza conversiones de tipo como se describe en §3.[3.9.1](#). (En la práctica, sin embargo, el operador de igualdad `==` está obsoleto en favor del operador de igualdad estricta `===`, que no realiza conversiones de tipo. Véase §4.[4.9.1](#) para más información sobre ambos operadores).

Las constantes y las variables le permiten utilizar nombres para referirse a valores en sus programas. Las constantes se declaran con `const` y las variables se declaran con `let` (o con `var` en el código JavaScript más antiguo). Las constantes y las variables de JavaScript son *no tipadas*: las declaraciones no especifican qué tipo de valores se asignarán. La declaración y asignación de variables se trata en §3.[3.10](#).

Como puedes ver en esta larga introducción, este es un capítulo muy amplio que explica muchos detalles fundamentales sobre cómo se representan y manipulan los datos en JavaScript. Empezaremos por sumergirnos en los detalles de los números y el texto de JavaScript.

## 3.2 Números

El tipo numérico principal de JavaScript, `Number`, se utiliza para representar números enteros y para aproximar números reales. JavaScript representa los números utilizando el formato de punto flotante de 64 bits definido por el IEEE 754,<sup>1</sup> lo que significa que puede representar números tan grandes como  $\pm 1,7976931348623157 \times 10^{308}$  y tan pequeños como  $\pm 5 \times 10^{-324}$ .

El formato numérico de JavaScript permite representar exactamente todos los números enteros entre -9.007.199.254.740.992 (-2<sup>53</sup>) y 9.007.199.254.740.992 (<sup>253</sup>), inclusive. Si utilizas valores enteros más grandes que esto, puedes perder precisión en los dígitos finales. Tenga en cuenta, sin embargo, que ciertas operaciones en JavaScript (como la indexación de matrices y los operadores a nivel de bits descritos en el [capítulo 4](#)) se realizan con enteros de 32 bits. Si necesita representar con exactitud enteros más grandes, consulte [§3.2.5.](#)

Cuando un número aparece directamente en un programa de JavaScript, se denomina *literal numérico*. JavaScript admite literales numéricos en varios formatos, como se describe en las siguientes secciones. Tenga en cuenta que cualquier literal numérico puede ir precedido de un signo menos (-) para que el número sea negativo.

### 3.2.1 Literales enteros

En un programa de JavaScript, un entero de base 10 se escribe como una secuencia de dígitos. Por ejemplo:

```
0  
3  
10000000
```

Además de los literales enteros de base 10, JavaScript reconoce los valores hexadecimales (base 16). Un literal hexadecimal comienza con 0x o 0X, seguido de una cadena de dígitos hexadecimales. Un dígito hexadecimal es uno de los dígitos del 0 al 9 o las letras de la a

(o A) a la f (o F), que representan los valores del 10 al 15. Estos son ejemplos de literales enteros hexadecimales:

```
0xff // => 255: (15*16 + 15)  
0xBADCAFE // => 195939070
```

En ES6 y posteriores, también se pueden expresar enteros en binario (base 2) u octal (base 8) utilizando los prefijos 0b y 0o (o 0B y 0O) en lugar de 0x:

```
0b10101 // => 21: (1*16 + 0*8 + 1*4 + 0*2 + 1*1)  
0o377 // => 255: (3*64 + 7*8 + 7*1)
```

### 3.2.2 Literales en coma flotante

Los literales de punto flotante pueden tener un punto decimal; utilizan la sintaxis tradicional de los números reales. Un valor real se representa como la parte integral del número, seguida de un punto decimal y la parte fraccionaria del número.

Los literales de punto flotante también pueden representarse utilizando la notación exponencial: un número real seguido de la letra e (o E), seguido de un signo opcional de más o menos, seguido de un exponente entero. Esta notación representa el número real multiplicado por 10 a la potencia del exponente.

Más sucintamente, la sintaxis es:

```
[dígitos][. dígitos][(E|e)[(+|-)]dígitos]
```

Por ejemplo:

```
3.14 2345.6789  
.3333333333333333  
6.02e23 //  $6.02 \times 10^{23}$   
1.4738223E-32 //  $1.4738223 \times 10^{-32}$ 
```

### SEPARADORES EN LOS LITERALES

Usted puede utilizar guiones bajos dentro de los literales numéricos para dividir los literales largos en trozos que sean más fáciles de leer:

```
dej_mil = 1_000_000_000; // El guión bajo como separador de miles.  
dej_bytes = 0x89EL_SISTEMA,DE // Como separador de bytes.  
dej_bits = 0b0001_1101_0111; // Como separador de nibbles.  
dej_fracción = 0.123_456_789; // También funciona en la parte fraccionaria.
```

En el momento de escribir este artículo, a principios de 2020, los guiones bajos en los literales numéricos todavía no estandarizado como parte de JavaScript. Pero están en las etapas avanzadas de la estandarización y son implementados por los principales navegadores y por Node.

### 3.2.3 Aritmética en JavaScript

Los programas de JavaScript trabajan con números utilizando los operadores aritméticos . que proporciona el lenguaje. Estos incluyen + para la suma, - para la resta, \* para la multiplicación, / para la división y % para el módulo (resto después de la división). ES2016 añade \*\* para la exponenciación. En [el capítulo 4 se](#) pueden encontrar todos los detalles sobre estos y otros operadores.

Además de estos operadores aritméticos básicos, JavaScript soporta operaciones matemáticas más complejas a través de un conjunto de funciones y constantes definidas como propiedades del objeto Math:

Math. `pow(2,53)` // => 9007199254740992: 2 a la potencia 53 Math. `round(.6)` // => 1.0: redondear al entero más cercano Math. `ceil(.6)` // => 1.0: redondear a un entero  
Math. `floor(.6)` // => 0.0: redondear a un entero  
Math. `abs(-5)` // => 5: valor absoluto  
Math. `max(x,y,z)` // Devuelve el mayor argumento  
Math. `min(x,y,z)` // Devuelve el argumento más pequeño  
Math. `random()` // Número pseudo-aleatorio x donde  $0 \leq x < 1.0$  Math. PI //  $\pi$ : circunferencia de un círculo / diámetro Math. E // e: La base del logaritmo natural  
Math. `sqrt(3)` // => 3\*\*0.5: la raíz cuadrada de 3  
Math. `pow(3, 1/3)` // => 3\*\*(1/3): la raíz cúbica de 3  
Math. `sin(0)` // Trigonometría: también Math.cos, Math.atan, etc.  
Math. `log(10)` // Logaritmo natural de 10  
Math. `log(100)/Math.LN10` // Logaritmo de base 10 de 100  
Math. `log(512)/Math.LN2` // Logaritmo de base 2 de 512  
Math. `exp(3)` // Math.E al cubo

ES6 define más funciones sobre el objeto Math:

Math. `cbrt(27)` // => 3: raíz cúbica  
Math. `hypot(3, 4)` // => 5: raíz cuadrada de la suma de los cuadrados de todos los argumentos  
Math. `log10(100)` // => 2: logaritmo de base-10  
Math. `log2(1024)` // => 10: Logaritmo de base 2  
Math. `log1p(x)` // Logaritmo natural de (1+x); preciso para x muy pequeño  
Math. `expm1(x)` // Math.exp(x)-1; la inversa de Math. `log1p()`  
Math. `sign(x)` // -1, 0, o 1 para argumentos <, ==, o > 0  
Math. `imul(2,3)` // => 6: multiplicación optimizada de enteros de 32 bits  
Math. `clz32(0xf)` // => 28: número de bits cero iniciales en un Entero de 32 bits  
Math. `trunc(3.9)` // => 3: convertir a un entero truncando la parte fraccionaria  
Math. `fround(x)` // Redondear al número flotante de 32 bits más cercano  
Math. `sinh(x)` // Seno hiperbólico. También Math. `cosh()`,  
Math. `tanh()`  
Math. `asinh(x)` // Arcoseno hiperbólico. También Math. `acosh()`,  
Math. `atanh()`

La aritmética en JavaScript no produce errores en casos de desbordamiento, subdesbordamiento o división por cero. Cuando el resultado de una operación numérica es mayor que el mayor número representable (desbordamiento), el resultado es un valor infinito especial, Infinito. Del mismo modo, cuando el valor absoluto de un valor negativo es mayor que el valor absoluto del mayor número negativo representable, el resultado es el infinito negativo, -Infinito. Los valores infinitos se comportan como es de esperar: sumar, restar, multiplicar o dividir por cualquier cosa da como resultado un valor infinito (posiblemente con el signo invertido).

El desbordamiento se produce cuando el resultado de una operación numérica está más cerca de cero que el menor número representable. En este caso, JavaScript devuelve 0. Si el desbordamiento se produce a partir de un número negativo, JavaScript devuelve un valor especial conocido como "cero

negativo". Este valor es casi completamente indistinguible del cero normal y los programadores de JavaScript rara vez necesitan detectarlo.

La división por cero no es un error en JavaScript: simplemente devuelve el infinito o el infinito negativo. Sin embargo, hay una excepción: el cero dividido entre cero no tiene un valor bien definido, y el resultado de esta operación es el valor especial no numérico, NaN. El valor NaN también surge si se intenta dividir infinito por infinito, sacar la raíz cuadrada de un número negativo o utilizar operadores aritméticos con operandos no numéricos que no pueden convertirse en números.

JavaScript predefine las constantes globales Infinity y NaN para mantener el valor del infinito positivo y del no-número, y estos valores también están disponibles como propiedades del objeto Number:

**Infinito** // Un número positivo demasiado grande para representar el número.

POSITIVE\_INFINITY // El mismo valor

**1/0** //=> Infinito

Número. **VALOR\_MAX \* 2** //=> Infinito; desbordamiento

-Infinito // Un número negativo demasiado grande para representarlo.

NEGATIVE\_INFINITY // El mismo valor

**-1/0** //=> -Infinito

-Número. **VALOR\_MAX \* 2** //=> -Infinito

**NaN** // El valor no numérico Número. **NaN** // El mismo valor, escrito de otra manera

**0/0** //=> NaN

**Infinito/Infinito** //=> NaN

Número. **MIN\_VALUE/2** //=> 0: underflow

-Número. **MIN\_VALUE/2** //=> -0: cero negativo

**-1/Infinito** //=> -0: también 0 negativo

-0

```
// Las siguientes propiedades de Number están definidas en ES6 Number. parseInt()  
// Igual que la función global parseInt() Number.parseFloat() // Igual que la función  
// global parseFloat() Number.isNaN(x) // ¿Es x el valor NaN?  
Número.isFinite(x) // ¿Es x un número y finito?  
Número.isInteger(x) // ¿Es x un número entero?  
Número.isSafeInteger(x) // ¿Es x un número entero -(2**53) < x < 2**53?  
Número.MIN_SAFE_INTEGER //=> -(2**53 - 1)  
Número.MAX_SAFE_INTEGER //=> 2**53 - 1  
Número.EPSILON //=> 2**-52: la menor diferencia entre números
```

El valor no numérico tiene una característica inusual en JavaScript: no se compara con ningún otro valor, incluido él mismo. Esto significa que no se puede escribir `x === NaN` para determinar si el valor de una variable `x` es `Nan`. En su lugar, debe escribir `x != x` o `Number.isNaN(x)`. Estas expresiones serán verdaderas si, y sólo si, `x` tiene el mismo valor que la constante global `Nan`.

La función global `isNaN()` es similar a `Number isNaN()`. Devuelve verdadero si su argumento es `Nan`, o si ese argumento es un valor no numérico que no puede convertirse en un número. La función relacionada `Number.isFinite()` devuelve `true` si su argumento es un número distinto de `Nan`, Infinito o -Infinito. La función global `isFinite()` devuelve `true` si su argumento es, o puede ser convertido en, un número finito.

El valor cero negativo también es algo inusual. Se compara igual (incluso utilizando la prueba de igualdad estricta de JavaScript) con el cero positivo, lo que significa que los dos valores son casi indistinguibles, excepto cuando se utiliza como un divisor:

```
let zero = 0; // Cero normal let negz = -0; // Cero negativo zero === negz //=>  
verdadero: cero y cero negativo son iguales  
1/cero === 1/negz //=> falso: Infinito y -Infinito no son iguales
```

### 3.2.4 Errores de punto flotante binario y de redondeo

Hay infinitos números reales, pero sólo un número finito de ellos (18.437.736.874.454.810.627, para ser exactos) puede representarse exactamente con el formato de coma flotante de JavaScript. Esto significa que cuando se trabaja con números reales en JavaScript, la representación del número será a menudo una aproximación del número real.

La representación de punto flotante IEEE-754 que utiliza JavaScript (y casi cualquier otro lenguaje de programación moderno) es una representación binaria, que puede representar exactamente fracciones como  $1/2$ ,  $1/8$  y  $1/1024$ . Desgraciadamente, las fracciones que utilizamos con más frecuencia (especialmente al realizar cálculos financieros) son fracciones decimales:  $1/10$ ,  $1/100$ , etc. Las representaciones binarias en coma flotante no pueden representar exactamente números tan simples como  $0,1$ .

Los números de JavaScript tienen mucha precisión y pueden aproximarse mucho a  $0,1$ . Pero el hecho de que este número no pueda representarse con exactitud puede dar lugar a problemas. Considere este código:

```
let x = .3 - .2; // treinta céntimos menos 20 céntimos let y = .2 - .1; // veinte céntimos menos 10 céntimos x === y //=> false: ¡los dos valores no son iguales!
x === .1 //=> falso: .3-.2 no es igual a .1 y === .1 // => verdadero: .2-.1 es igual a .1
```

Debido al error de redondeo, la diferencia entre las aproximaciones de  $.3$  y  $.2$  no es exactamente la misma que la diferencia entre las aproximaciones de  $.2$  y  $.1$ . Es importante entender que este problema no es específico de JavaScript: afecta a cualquier lenguaje

de programación que utilice números binarios en coma flotante. Además, hay que tener en cuenta que los valores x e y en el código que se muestra aquí están *muy próximos entre sí* y al valor correcto. Los valores calculados son adecuados para casi cualquier propósito; el problema sólo surge cuando intentamos comparar valores para la igualdad.

Si estas aproximaciones de punto flotante son problemáticas para sus programas, considere el uso de enteros escalados. Por ejemplo, puede manipular los valores monetarios como centavos enteros en lugar de dólares fraccionarios.

### 3.2.5 Números enteros de precisión arbitraria con BigInt

Una de las características más nuevas de JavaScript, definida en ES2020, es un nuevo tipo numérico conocido como BigInt. A principios de 2020, se ha implementado en Chrome, Firefox, Edge y Node, y hay una implementación en curso en Safari. Como su nombre indica, BigInt es un tipo numérico cuyos valores son enteros. El tipo se añadió a JavaScript principalmente para permitir la representación de enteros de 64 bits, que son necesarios para la compatibilidad con muchos otros lenguajes de programación y API. Pero los valores de BigInt pueden tener miles o incluso millones de dígitos, en caso de que necesites trabajar con números tan grandes. (Tenga en cuenta, sin embargo, que las implementaciones de BigInt no son adecuadas para la criptografía porque no intentan evitar los ataques de tiempo).

Los literales BigInt se escriben como una cadena de dígitos seguida de una letra n minúscula. Por defecto, están en base 10, pero se pueden

utilizar los prefijos 0b, 0o y 0x para BigInts binarios, octales y hexadecimales:

```
1234n // Un literal BigInt no tan grande  
0b111111n // Un BigInt binario  
0o7777n // Un BigInt octal  
0x8000000000000000n // => 2n**63n: Un entero de 64 bits
```

Puede utilizar BigInt() como función para convertir números o cadenas regulares de JavaScript en valores BigInt:

```
BigInt(Número. MAX_SAFE_INTEGER) // => 9007199254740991n let string = "1" +  
"0". repeat(100); // 1 seguido de 100 ceros. BigInt(cadena) // => 10n**100n: un  
googol
```

La aritmética con valores BigInt funciona como la aritmética con números normales de JavaScript, excepto que la división elimina cualquier resto y redondea hacia abajo (hacia el cero):

```
1000n + 2000n // => 3000n  
3000n - 2000n // => 1000n  
2000n * 3000n // => 6000000n  
3000n / 997n // => 3n: el cociente es 3  
3000n % 997n // => 9n: y el resto es 9 (2n ** 131071n) - 1n // Un primo de Mersenne  
con 39457 cifras decimales
```

Aunque los operadores estándar +, -, \*, /, % y \*\* funcionan con BigInt, es importante entender que no se pueden mezclar operandos de tipo BigInt con operandos numéricos normales. Esto puede parecer confuso al principio, pero hay una buena razón para ello. Si un tipo numérico fuera más general que el otro, sería fácil definir la aritmética sobre los operandos mezclados para devolver simplemente un valor del tipo más general. Pero ningún tipo es más general que el otro: BigInt puede representar valores extraordinariamente grandes, por lo que es más general que los números normales. Pero BigInt sólo puede representar números

enteros, lo que hace que el tipo de número regular de JavaScript sea más general. No hay forma de evitar este problema, por lo que JavaScript lo esquiva simplemente no permitiendo operandos mixtos a los operadores aritméticos.

Los operadores de comparación, por el contrario, sí funcionan con tipos numéricos mixtos (pero véase [§3.9.1](#) para saber más sobre la diferencia entre `==` y `===`):

```
1 < 2n // => true
2 > 1n // => true
0 == 0n // => verdadero
0 === 0n // => false: el === comprueba también la igualdad de tipos
```

Los operadores bit a bit (descritos en [§4.8.3](#)) generalmente funcionan con operandos BigInt. Sin embargo, ninguna de las funciones del objeto Math acepta operandos BigInt.

### 3.2.6 Fechas y horarios

JavaScript define una clase Date sencilla para representar y manipular los números que representan fechas y horas. Las fechas de JavaScript son objetos, pero también tienen una representación numérica como *marca de tiempo* que especifica el número de milisegundos transcurridos desde el 1 de enero de 1970:

```
let timestamp = Date.now(); // La hora actual como marca de tiempo (un número).
let now = new Date(); // La hora actual como objeto Date.
let ms = now.getTime(); // Convertir a una marca de tiempo en milisegundos.
let iso = now.toISOString(); // Convertir a una cadena en formato estándar.
```

La clase Date y sus métodos se tratan en detalle en [§11.4](#). Pero volveremos a ver los objetos Date en [§3.9.3](#) cuando examinemos los detalles de las conversiones de tipo de JavaScript.

## 3.3 Texto

El tipo de JavaScript para representar texto es la *cadena*. Una cadena es una secuencia ordenada e inmutable de valores de 16 bits, cada uno de los cuales suele representar un carácter Unicode. La *longitud* de una cadena es el número de valores de 16 bits que contiene. Las cadenas de JavaScript (y sus matrices) utilizan una indexación basada en cero: el primer valor de 16 bits está en la posición 0, el segundo en la posición 1, y así sucesivamente. La *cadena vacía* es la cadena de longitud 0. JavaScript no tiene un tipo especial que represente un solo elemento de una cadena. Para representar un único valor de 16 bits, basta con utilizar una cadena que tenga una longitud de 1.

### CARACTERES, PUNTOS DE CÓDIGO Y CADENAS DE JAVASCRIPT

JavaScript utiliza la codificación UTF-16 del conjunto de caracteres Unicode, y las cadenas de JavaScript son secuencias de valores de 16 bits sin signo. Los caracteres Unicode más utilizados (los del "plano básico multilingüe") tienen puntos de código que caben en 16 bits y pueden representarse mediante un elemento de una cadena. Los caracteres Unicode cuyos puntos de código no caben en 16 bits se codifican utilizando las reglas de

UTF-16 como una secuencia (conocida como *pares sustituto*) de dos valores de 16 bits. Esto significa que una cadena JavaScript de longitud 2 (dos valores de 16 bits) podría representar un solo carácter Unicode.

```
dejeuro = "€";
dejamor= "♥";
euro.longitud //=> 1: este carácter tiene un elemento de 16 bits
amorlongitud //=> 2: Codificación UTF16"\ud83d\udc99"
```

La mayoría de los métodos de manipulación de cadenas definidos por JavaScript operan con valores de 16 bits. No tratan especialmente los pares sustitutos, no realizan ninguna normalización de la cadena y ni siquiera aseguran que una cadena está bien formada UTF-16.

E En ES6, sin embargo, las `for...of` y si se utiliza el `for...in` para iterar sobre una cadena, es que iterará los caracteres reales de la cadena, no los valores de 16 bits.

### 3.3.1 Literales de cadena

Para incluir una cadena en un programa JavaScript, basta con encerrar los caracteres de la cadena dentro de un par de comillas simples o dobles o de puntos suspensivos (' o " o `'). Los caracteres de las comillas dobles y los puntos suspensivos pueden estar contenidos en cadenas delimitadas por caracteres de comillas simples, y lo mismo ocurre con las cadenas delimitadas por comillas dobles y puntos suspensivos. A continuación se muestran ejemplos de literales de cadena:

```
"" // La cadena vacía: tiene cero caracteres
"pruebas
"3.14"
'name="myform"
"¿No prefieres el libro de O'Reilly?"
"π es la relación entre la circunferencia y el radio de un círculo"
"Ella dijo 'hola'", dijo.
```

Las cadenas delimitadas con marcas son una característica de ES6, y permiten que las expresiones de JavaScript sean incrustadas dentro (o *interpoladas*) de la cadena literal. Esta sintaxis de interpolación de expresiones se trata en [§3.3.4](#).

Las versiones originales de JavaScript requerían que los literales de cadena se escribieran en una sola línea, y es común ver código JavaScript que crea cadenas largas concatenando cadenas de una sola línea con el operador +. Sin embargo, a partir de ES5, se puede dividir una cadena literal en varias líneas terminando cada línea menos la última con una barra invertida (\). Ni la barra invertida ni el terminador de línea que la sigue forman parte del literal de cadena. Si necesita incluir un carácter de nueva línea en un literal de cadena entre comillas simples o dobles, utilice la secuencia de caracteres \n (documentada en la siguiente sección). La sintaxis de backtick de ES6 permite dividir

las cadenas en varias líneas, y en este caso, los terminadores de línea forman parte del literal de la cadena:

```
// Una cadena que representa 2 líneas escritas en una línea:  
'dos\nlíneas'
```

```
// Una cadena de una línea escrita en 3 líneas:  
"una línea  
larga"
```

```
// Una cadena de dos líneas escrita en dos líneas:  
'el carácter de nueva línea al final de esta línea se incluye  
literalmente en esta cadena'
```

Tenga en cuenta que cuando utilice comillas simples para delimitar sus cadenas, debe tener cuidado con las contracciones y posesivos ingleses, como *can't* y *O'Reilly's*. Dado que el apóstrofe es el mismo que el carácter de comillas simples, debe utilizar el carácter de barra invertida (\) para "escapar" de cualquier apóstrofe que aparezca en las cadenas entre comillas simples (los escapes se explican en la siguiente sección).

En la programación JavaScript del lado del cliente, el código JavaScript puede contener cadenas de código HTML, y el código HTML puede contener cadenas de código JavaScript. Al igual que JavaScript, HTML utiliza comillas simples o dobles para delimitar sus cadenas. Por lo tanto, al combinar JavaScript y HTML, es una buena idea utilizar un estilo de comillas para JavaScript y el otro estilo para HTML. En el siguiente ejemplo, la cadena "Gracias" está entre comillas simples dentro de una expresión JavaScript, que a su vez está entre comillas dobles dentro de un atributo HTML event-handler:

```
<button onclick="alert('Gracias')"> Hazme clic</button>
```

### 3.3.2 Secuencias de escape en literales de cadena

El carácter de barra invertida (\) tiene un propósito especial en las cadenas de JavaScript.

Combinado con el carácter que le sigue, representa un carácter que no es representable de otra manera dentro de la cadena. Por ejemplo, \n es una *secuencia de escape* que representa un carácter de nueva línea.

Otro ejemplo, mencionado anteriormente, es la secuencia de escape \N que representa el carácter de comilla simple (o apóstrofe). Esta secuencia de escape es útil cuando se necesita incluir un apóstrofe en una cadena literal que está contenida entre comillas simples.

Puede ver por qué se llaman secuencias de escape: la barra invertida le permite escapar de la interpretación habitual del carácter de comillas simples. En lugar de utilizarla para marcar el final de la cadena, se utiliza como un apóstrofe:

"Tienes razón, no puede ser una cita

La Tabla 3-1 enumera las secuencias de escape de JavaScript y los caracteres que representan. Tres secuencias de escape son genéricas y pueden utilizarse para representar cualquier carácter especificando su código de carácter Unicode como número hexadecimal. Por ejemplo, la secuencia \xA9 representa el símbolo de copyright, que tiene la codificación Unicode dada por el número hexadecimal A9. Del mismo modo, el escape \u representa un carácter Unicode arbitrario especificado por cuatro dígitos hexadecimales o de uno a cinco dígitos cuando los dígitos están encerrados entre llaves: \u03c0 representa el carácter π, por ejemplo, y \u{1f600} representa el emoji "cara sonriente".

*Tabla 3-1. Secuencias de escape de JavaScript*

**Secuencia**

**Personaje representado**

\0 El carácter NUL (\u0000)

\b Retroceso (\u0008)

\t Ficha horizontal (\u0009)

\n Nueva línea (\u000A)

\v Pestaña vertical (\u000B)

\f Alimentación del formulario (\u000C)

\r Retorno de carro (\u000D)

\\" Comillas dobles (\u0022)

\' Apóstrofe o comilla simple (\u0027)

\\\ Barra invertida (\u005C)

\N - La  
vida de  
los niños El carácter Unicode especificado por los dos dígitos hexadecimales *nn*  
en la  
escuela

|         |  |
|---------|--|
| \N - nn | El carácter Unicode especificado por los cuatro dígitos hexadecimales nnnn |
|---------|--|

|         |  |
|---------|--|
| \N-u{n} | El carácter Unicode especificado por el punto de código n, donde n es de uno a seis dígitos hexadecimales entre 0 y 10FFFF (ES6) |
|---------|--|

Si el carácter \N precede a cualquier otro carácter distinto de los indicados en

Tabla 3-1, la barra invertida simplemente se ignora (aunque las futuras versiones del lenguaje pueden, por supuesto, definir nuevas secuencias de escape). Por ejemplo, \# es lo mismo que #. Finalmente, como se ha señalado anteriormente, ES5 permite una barra invertida antes de un salto de línea para romper una cadena literal a través de múltiples líneas.

### 3.3.3 Trabajar con cadenas

Una de las características integradas de JavaScript es la capacidad de *concatenar* cadenas. Si utilizas el operador + con números, los suma. Pero si usas este operador con cadenas, las une añadiendo la segunda a la primera. Por ejemplo:

```
let msg = "Hola, " + "mundo"; // Produce la cadena "Hola, mundo"
let greeting = "Bienvenido a mi blog," + " " + nombre;
```

Las cadenas pueden compararse con los operadores estándar de igualdad === y desigualdad !==: dos cadenas son iguales si y sólo si están formadas por exactamente la misma secuencia de valores de 16 bits. Las cadenas también pueden compararse con los operadores <, <=, > y >=. La comparación de cadenas se realiza simplemente comparando los valores de 16 bits. (Para una comparación y ordenación de cadenas más robusta de localeaware, véase §11.7.3.)

Para determinar la longitud de una cadena -el número de valores de 16 bits que contiene- utilice la propiedad length de la cadena:

#### s. longitud

Además de esta propiedad de longitud, JavaScript proporciona una rica API para trabajar con cadenas:

```
let s = "Hola, mundo"; // Comienza con algún texto.

// Obtención de porciones de una cadena
s.substring(1,4) // => "ell": los caracteres 2, 3 y 4.
s.slice(1,4) // => "ell": lo mismo
s.slice(-3) // => "rld": últimos 3 caracteres
s.split(", ") // => ["Hola", "mundo"]: split en la cadena delimitadora

// Búsqueda de una cadena
s.indexOf("l") // => 2: posición de la primera letra l
s.indexOf("l", 3) // => 3: posición de la primera "l" en o después de 3
s.indexOf("zz") // => -1: s no incluye la subcadena "zz"
s.lastIndexOf("l") // => 10: posición de la última letra l

// Funciones de búsqueda booleana en ES6 y posteriores
s.startsWith("Hell") // => true: la cadena empieza por estos
s.endsWith("!") // => false: s no termina con eso
s.includes("or") // => true: s incluye la subcadena "or"

// Crear versiones modificadas de una cadena
s.replace("llo", "ya") // => "Heya, world"
s.toLowerCase() // => "hola, mundo"
s.toUpperCase() // => "HOLA, MUNDO"
s.normalize() // Normalización Unicode NFC: ES6
s.normalize("NFD") // Normalización NFD. También "NFKC", "NFKD"

// Inspección de caracteres individuales (16 bits) de una cadena
s.charAt(0) // => "H": el primer carácter
s.charAt(s.length-1) // => "d": el último carácter
s.charCodeAt(0) // => 72: número de 16 bits en la posición especificada
s.codePointAt(0) // => 72: ES6, funciona para puntos de código >
```

16 bits

// Funciones de relleno de cadenas en ES2017 "x". padStart(3) // => "x": añade espacios a la izquierda hasta una longitud de 3 "x". padEnd(3) // => "x": añade espacios a la derecha hasta una longitud de 3 "x". padStart(3, "\*") // => "\*\*\*x": añade estrellas a la izquierda hasta una longitud de 3 "x". padEnd(3, "-") // => "x--": añade guiones a la derecha hasta una longitud de 3

// Funciones de recorte de espacios. trim() es ES5; otras ES2019 " test ". trim() // => "test": eliminar espacios al principio y al final " test ". trimStart() // => "test": eliminar espacios a la izquierda.

También trimLeft " test ". trimEnd() // => " test": elimina los espacios a la derecha. También trimRight

// Varios métodos de cadena

s. concat("!") // => "¡Hola, mundo!": sólo hay que usar el operador + en su lugar "**<>**". repeat(5) // => "<><><><>": concatena n copias. ES6

Recuerda que las cadenas son inmutables en JavaScript. Métodos como replace() y toUpperCase() devuelven nuevas cadenas: no modifican la cadena sobre la que se invocan.

Las cadenas también pueden tratarse como matrices de sólo lectura, y se puede acceder a caracteres individuales (valores de 16 bits) de una cadena utilizando corchetes en lugar del método charAt():

```
let s = "hola, mundo"; s[0] // => "h" s[s.length-1] // => "d"
```

### 3.3.4 Literales de las plantillas

En ES6 y en versiones posteriores, los literales de cadena pueden delimitarse con puntos suspensivos:

```
let s = `hola mundo`;
```

Sin embargo, se trata de algo más que otra sintaxis de literales de cadena, ya que estos *literales de plantilla* pueden incluir expresiones JavaScript arbitrarias. El valor final de un literal de cadena en barras invertidas se calcula evaluando cualquier expresión incluida, convirtiendo los valores de esas expresiones en cadenas y combinando esas cadenas calculadas con los caracteres literales dentro de las barras invertidas:

```
let name = "Bill"; let greeting = `Hola ${ name }`; // greeting == "Hola Bill".
```

Todo lo que está entre el \${ y el } correspondiente se interpreta como un expresión de JavaScript. Todo lo que está fuera de las llaves es un texto literal de cadena normal. La expresión dentro de las llaves se evalúa y luego se convierte en una cadena y se inserta en la plantilla, reemplazando el signo de dólar, las llaves y todo lo que hay entre ellas.

Un literal de plantilla puede incluir cualquier número de expresiones. Puede utilizar cualquiera de los caracteres de escape que las cadenas normales pueden utilizar, y puede abarcar cualquier número de líneas, sin necesidad de escapes especiales. El siguiente literal de plantilla incluye cuatro expresiones JavaScript, una secuencia de escape Unicode y al menos cuatro nuevas líneas (los valores de las expresiones también pueden incluir nuevas líneas):

```
let MensajeError = `\\Nmensaje de error
\u2718 Fallo de la prueba en ${filename}:${linenumber}:
${exception.message}
Rastreo de pila:
${exception.stack}
`;
```

La barra invertida al final de la primera línea escapa a la nueva línea inicial para que la cadena resultante comience con el carácter Unicode **X** (\u2718) en lugar de una nueva línea.

## LITERALES DE PLANTILLAS ETIQUETADAS

Una característica poderosa pero menos utilizada de los literales de plantilla es que, si el nombre de una función (o "etiqueta") viene justo antes de la marca de apertura, entonces el texto y los valores de las expresiones dentro del literal de plantilla se pasan a la función. El valor de este "literal de plantilla etiquetado" es el valor de retorno de la función. Esto podría utilizarse, por ejemplo, para aplicar el escape HTML o SQL a los valores antes de sustituirlos en el texto.

ES6 tiene una función de etiqueta incorporada: `String.raw()`. Devuelve el texto dentro de las barras invertidas sin procesar las barras invertidas:

```
\n`.length // => 1: la cadena tiene un solo carácter de nueva línea String.raw`\n`.  
length // => 2: un carácter de barra invertida y la letra n
```

Tenga en cuenta que aunque la parte de la etiqueta de un literal de plantilla etiquetado es una función, no se utilizan paréntesis en su invocación. En este caso tan específico, los caracteres de la palanca de cambios sustituyen a los paréntesis de apertura y cierre.

La posibilidad de definir tus propias funciones de etiqueta de plantilla es una característica poderosa de JavaScript. Estas funciones no necesitan devolver cadenas, y pueden usarse como constructores, como si se definiera una nueva sintaxis literal para el lenguaje. Veremos un ejemplo en [§14.5](#).

### 3.3.5 Comparación de patrones

JavaScript define un tipo de datos conocido como *expresión regular* (o RegExp) para describir y comparar patrones en cadenas de texto. Las RegExp no son uno de los tipos de datos fundamentales de JavaScript, pero tienen una sintaxis literal como la de los números y las cadenas, por lo que a veces parece que son fundamentales. La gramática de las expresiones regulares literales es compleja y la API que definen no es trivial. Se documentan en detalle en [§11.3](#). Sin embargo, debido a que las RegExps son poderosas y comúnmente usadas para el procesamiento de texto, esta sección proporciona una breve visión general.

El texto entre un par de barras constituye un literal de expresión regular. La segunda barra del par también puede ir seguida de una o más letras, que modifican el significado del patrón. Por ejemplo:

```
/^HTML/; // Coincidan con las letras H T M L al principio de una cadena /[1-9][0-9]*/; // Coincidan con un dígito no nulo, seguido de cualquier número de dígitos /\bjavascript\b/i; // Coincidan con "javascript" como una palabra, que distingue entre mayúsculas y minúsculas
```

Los objetos RegExp definen una serie de métodos útiles, y las cadenas también tienen métodos que aceptan argumentos RegExp. Por ejemplo:

```
let text = "probando: 1, 2, 3"; // Ejemplo de texto let pattern = /\d+/g; // Coincide con todas las instancias de uno o más dígitos
pattern.test(text) // => true: existe una coincidencia text.search(pattern) // => 9: posición de la primera coincidencia text.match(pattern) // => ["1", "2", "3"]: matriz de todas las coincidencias text.replace(pattern, "#") // => "testing: #, #, #" text.
split(/\D+/) // => [", "1", "2", "3"]: división en los no dígitos
```

## 3.4 Valores booleanos

Un valor booleano representa la verdad o la falsedad, encendido o apagado, sí o no. Sólo hay dos valores posibles de este tipo. Las palabras reservadas `true` y `false` evalúan a estos dos valores.

Los valores booleanos suelen ser el resultado de las comparaciones que se realizan en los programas de JavaScript. Por ejemplo:

```
a === 4
```

Este código comprueba si el valor de la variable `a` es igual al número 4. Si lo es, el resultado de esta comparación es el valor booleano `true`. Si `a` no es igual a 4, el resultado de la comparación es `false`.

Los valores booleanos se utilizan habitualmente en las estructuras de control de JavaScript. Por ejemplo, la sentencia `if/else` en JavaScript realiza una acción si un valor booleano es verdadero y otra acción si el valor es falso. Normalmente se combina una comparación que crea un valor booleano directamente con una sentencia que lo utiliza. El resultado es así:

```
if (a === 4) { b = b +  
1; } else { a = a + 1;  
}
```

Este código comprueba si `a` es igual a 4. Si es así, añade 1 a `b`; en caso contrario, añade 1 a `a`.

Como veremos en §3.9, cualquier valor de JavaScript puede convertirse en un valor booleano. Los siguientes valores se convierten a `false`, y por lo tanto funcionan como `false`:

**indefinido nulo**  
0  
-0  
**NaN**  
"" // la cadena vacía

Todos los demás valores, incluyendo todos los objetos (y arrays) se convierten y funcionan como true. false, y los seis valores que se convierten a él, se llaman a veces valores *falsos*, y todos los demás valores se llaman *truthy*. Cada vez que JavaScript espera un valor booleano, un valor falso funciona como falso y un valor verdadero funciona como verdadero.

Como ejemplo, supongamos que la variable o contiene un objeto o el valor null. Puede comprobar explícitamente si o no es nulo con una sentencia if como la siguiente:

**if (o !== null)** ...

El operador no-igual !== compara o con null y evalúa como verdadero o falso. Pero puede omitir la comparación y confiar en el hecho de que null es falso y los objetos son verdaderos:

**si (o)** ...

En el primer caso, el cuerpo del if se ejecutará sólo si o no es null. El segundo caso es menos estricto: se ejecutará el cuerpo del if sólo si o no es falso o cualquier valor falso (como null o undefined). Qué sentencia if es apropiada para su programa depende realmente de los valores que espera que se asignen a o. Si necesita distinguir entre null y 0 y "", entonces debe utilizar una comparación explícita.

Los valores booleanos tienen un método `toString()` que puede utilizar para convertirlos en las cadenas "verdadero" o "falso", pero no

tienen ningún otro método útil. A pesar de la trivialidad de la API, hay tres operadores booleanos importantes.

El operador `&&` realiza la operación booleana AND. Se evalúa con un valor verdadero si y sólo si ambos operandos son verdaderos; en caso contrario, se evalúa con un valor falso. El operador `||` es la operación booleana OR: se evalúa con un valor verdadero si uno (o ambos) de sus operandos es verdadero y se evalúa con un valor falso si ambos operandos son falsos. Por último, el operador unario `!` realiza la operación booleana NOT: se evalúa como verdadero si su operando es falso y se evalúa como falso si su operando es verdadero. Por ejemplo:

```
if ((x === 0 && y === 0) || !(z === 0)) { // x e y son ambos cero o z es  
// distinto de cero  
}
```

Los detalles completos de estos operadores se encuentran en [§4.10](#).

## 3.5 nulos e indefinidos

`null` es una palabra clave del lenguaje que se evalúa a un valor especial que se suele utilizar para indicar la ausencia de un valor. El uso del operador `typeof` sobre `null` devuelve la cadena "object", lo que indica que `null` puede considerarse como un valor especial de objeto que indica "no object". En la práctica, sin embargo, `null` se considera normalmente como el único miembro de su propio tipo, y puede utilizarse para indicar "ningún valor" para números y cadenas, así como para objetos. La mayoría de los lenguajes de programación tienen un equivalente al `null` de JavaScript: puede que lo conozca como `NULL`, `nil` o `None`.

JavaScript también tiene un segundo valor que indica la ausencia de valor. El valor indefinido representa un tipo de ausencia más profundo. Es el valor de las variables que no han sido inicializadas y el valor que se obtiene cuando se consulta el valor de una propiedad de un objeto o de un elemento de un array que no existe. El valor indefinido es también el valor de retorno de las funciones que no devuelven explícitamente un valor y el valor de los parámetros de las funciones para las que no se pasa ningún argumento. `undefined` es una constante global predefinida (no una palabra clave del lenguaje como `null`, aunque esto no es una distinción importante en la práctica) que se inicializa al valor indefinido. Si se aplica el operador `typeof` al valor indefinido, éste devuelve "`undefined`", indicando que este valor es el único miembro de un tipo especial.

A pesar de estas diferencias, tanto `null` como `undefined` indican una ausencia de valor y a menudo pueden utilizarse indistintamente. El operador de igualdad `==` los considera iguales. (Utilice el operador de igualdad estricta `==` para distinguirlos.) Ambos son valores falsos: se comportan como `false` cuando se requiere un valor booleano. Ni `null` ni `undefined` tienen propiedades o métodos. De hecho, utilizar `.` o `[]` para acceder a una propiedad o método de estos valores provoca un `TypeError`.

Considero que `undefined` representa una ausencia de valor a nivel de sistema, inesperada o similar a un error, y `null` representa una ausencia de valor a nivel de programa, normal o esperada. Evito usar `null` e indefinido cuando puedo, pero si necesito asignar uno de estos valores a una variable o propiedad o pasar o devolver uno de estos valores a o desde una función, normalmente uso `null`. Algunos

programadores se esfuerzan por evitar null por completo y utilizan undefined en su lugar siempre que pueden.

## 3.6 Símbolos

Los símbolos se introdujeron en ES6 para servir como nombres de propiedades que no sean cadenas. Para entender Symbols, es necesario saber que el tipo fundamental Object de JavaScript es una colección desordenada de propiedades, donde cada propiedad tiene un nombre y un valor. Los nombres de las propiedades son típicamente (y hasta ES6, eran exclusivamente) cadenas. Pero en ES6 y posteriores, los Symbols también pueden servir para este propósito:

```
let strname = "string name"; // Una cadena para usar como nombre de propiedad let
symname = Symbol("propname"); // Un símbolo para usar como nombre de
propiedad typeof strname // => "string": strname es una cadena typeof symname // => "symbol": symname es un símbolo let o = {}; // Crear un nuevo objeto o[strname] = 1; // Definir una propiedad con un nombre de cadena o[symname] = 2; // Definir una propiedad con nombre de símbolo o[strname] // => 1: acceder a la propiedad con nombre de cadena o[symname] // => 2: acceder a la propiedad con nombre de símbolo
```

El tipo Symbol no tiene una sintaxis literal. Para obtener un valor Symbol, se llama a la función Symbol(). Esta función nunca devuelve el mismo valor dos veces, incluso cuando se llama con el mismo argumento. Esto significa que si se llama a Symbol() para obtener un valor Symbol, se puede utilizar con seguridad ese valor como nombre de propiedad para añadir una nueva propiedad a un objeto y no hay que preocuparse por si se sobrescribe una propiedad existente con el mismo nombre. Del mismo modo, si utiliza nombres de propiedades simbólicas y no comparte esos

símbolos, puede estar seguro de que otros módulos de código en su programa no sobrescribirán accidentalmente sus propiedades.

En la práctica, Symbols sirve como mecanismo de extensión del lenguaje. Cuando ES6 introdujo el bucle `for/of` ([§5.4.4](#)) y los objetos iterables ([Capítulo 12](#)), necesitó definir un método estándar que las clases pudieran implementar para hacerse iterables. Pero estandarizar cualquier nombre de cadena particular para este método iterador habría roto el código existente, así que se utilizó un nombre simbólico en su lugar. Como veremos en el [Capítulo 12](#), `Symbol.iterator` es un valor `Symbol` que puede utilizarse como nombre de método para hacer un objeto iterable.

La función `Symbol()` toma un argumento de cadena opcional y devuelve un valor único de `Symbol`. Si proporciona un argumento de cadena, esa cadena se incluirá en la salida del método `toString()` del `Symbol`. Tenga en cuenta, sin embargo, que llamar a `Symbol()` dos veces con la misma cadena produce dos valores `Symbol` completamente diferentes.

```
deje s = Symbol("sym_x");
s. toString() // => "Symbol(sym_x)"
```

`toString()` es el único método interesante de las instancias de `Symbol`. Sin embargo, hay otras dos funciones relacionadas con `Symbol` que deberías conocer. A veces, cuando se usan `Symbols`, se quiere mantenerlos privados para el propio código, de modo que se tiene la garantía de que sus propiedades nunca entrarán en conflicto con las propiedades usadas por otro código. Otras veces, sin embargo, puede querer definir un valor de `Symbol` y compartirlo ampliamente con otro código. Este sería el caso, por ejemplo, si estuvieras definiendo

algún tipo de extensión en la que quisieras que otro código pudiera participar, como con el mecanismo `Symbol.iterator` descrito anteriormente.

Para servir a este último caso de uso, JavaScript define un registro global de `Symbol`. La función `Symbol.for()` toma un argumento de cadena y devuelve un valor de `Symbol` que está asociado a la cadena que se pasa. Si no hay ningún `Symbol` asociado a esa cadena, se crea uno nuevo y se devuelve; en caso contrario, se devuelve el `Symbol` ya existente. Es decir, la función `Symbol.for()` es completamente diferente a la función `Symbol()`: `Symbol()` nunca devuelve el mismo valor dos veces, pero `Symbol.for()` siempre devuelve el mismo valor cuando se llama con la misma cadena. La cadena pasada a `Symbol.for()` aparece en la salida de `toString()` para el `Symbol` devuelto, y también puede ser recuperada llamando a `Symbol.keyFor()` sobre el `Symbol` devuelto.

```
let s = Symbol.for("shared"); let t =  
Symbol.for("shared"); s === t //=> true  
s.toString() //=> "Symbol(shared)"  
Symbol.keyFor(t) //=> "shared"
```

## 3.7 El objeto global

En las secciones anteriores se han explicado los tipos y valores primitivos de JavaScript. Los tipos de objetos -objetos, arrays y funciones- se tratan en capítulos propios más adelante en este libro. Pero hay un valor de objeto muy importante que debemos cubrir ahora. El *objeto global* es un objeto regular de JavaScript que sirve para un propósito muy importante: las propiedades de este objeto son los identificadores definidos globalmente que están disponibles para un programa de JavaScript. Cuando el intérprete de JavaScript se inicia (o cada vez que un navegador web carga una nueva página),

crea un nuevo objeto global y le da un conjunto inicial de propiedades que definen:

- Constantes globales como undefined, Infinity y NaN
- Funciones globales como isNaN(), parseInt() ([§3.9.2](#)) y eval() ([§4.12](#))
- Funciones constructoras como Date(), RegExp(), String(), Object(), y Array() ([§3.9.2](#))
- Objetos globales como Math y JSON ([§6.8](#))

Las propiedades iniciales del objeto global no son palabras reservadas, pero merecen ser tratadas como si lo fueran. Este capítulo ya ha descrito algunas de estas propiedades globales. La mayoría de las demás se tratarán en otras partes de este libro.

En Node, el objeto global tiene una propiedad llamada global cuyo valor es el propio objeto global, por lo que siempre puedes referirte al objeto global por el nombre global en los programas Node.

En los navegadores web, el objeto Window sirve como objeto global para todo el código JavaScript contenido en la ventana del navegador que representa. Este objeto global Window tiene una propiedad window autorreferencial que puede utilizarse para referirse al objeto global. El objeto Window define las propiedades globales principales, pero también define bastantes otras globales que son específicas de los navegadores web y de JavaScript del lado del cliente. Los hilos de trabajo de la web ([§15.13](#)) tienen un objeto global diferente al de la ventana a la que están asociados. El código de un worker puede referirse a su objeto global como self.

ES2020 define finalmente `globalThis` como la forma estándar de referirse al objeto global en cualquier contexto. A principios de 2020, esta característica ha sido implementada por todos los navegadores modernos y por Node.

## 3.8 Valores primitivos inmutables y referencias a objetos mutables

Hay una diferencia fundamental en JavaScript entre los valores primitivos (indefinidos, nulos, booleanos, números y cadenas) y los objetos (incluyendo arrays y funciones). Los primitivos son inmutables: no hay forma de cambiar (o "mutar") un valor primitivo. Esto es obvio en el caso de los números y los booleanos: ni siquiera tiene sentido cambiar el valor de un número. Sin embargo, no es tan obvio en el caso de las cadenas. Dado que las cadenas son como matrices de caracteres, es de esperar que se pueda modificar el carácter en cualquier índice especificado. De hecho, JavaScript no lo permite, y todos los métodos de cadena que parecen devolver una cadena modificada están, de hecho, devolviendo un nuevo valor de cadena. Por ejemplo:

```
let s = "hola"; // Comienza con un texto en minúsculas
s.toUpperCase(); // Devuelve "HELLO", pero no altera s // => "hello": la cadena
original no ha cambiado
```

Las primitivas también se comparan *por valor*: dos valores son iguales sólo si tienen el mismo valor. Esto parece circular para los números, los booleanos, los nulos y los indefinidos: no hay otra forma de compararlos. Sin embargo, no es tan obvio para las cadenas. Si se comparan dos valores de cadena distintos, JavaScript los considera

iguales si, y sólo si, tienen la misma longitud y si el carácter de cada índice es el mismo.

Los objetos son diferentes a las primitivas. En primer lugar, son *mutables*: sus valores pueden cambiar:

```
let o = { x: 1 }; // Empezar con un objeto
o.x = 2; // Mutarlo cambiando el valor de una propiedad
o.y = 3; // Mutarlo de nuevo añadiendo una nueva propiedad

let a = [1,2,3]; // Los arrays también son mutables a[0] = 0; // Cambiar el valor de un elemento del array a[3] = 4; // Añadir un nuevo elemento del array
```

Los objetos no se comparan por valor: dos objetos distintos no son iguales aunque tengan las mismas propiedades y valores. Y dos matrices distintas no son iguales aunque tengan los mismos elementos en el mismo orden:

```
let o = {x: 1}, p = {x: 1}; // Dos objetos con las mismas propiedades o === p // => false: los objetos distintos nunca son iguales let a = [], b = []; // Dos matrices distintas y vacías a === b // => false: las matrices distintas nunca son iguales
```

Los objetos se denominan a veces *tipos de referencia* para distinguirlos de los tipos primitivos de JavaScript. Utilizando esta terminología, los valores de los objetos son *referencias*, y decimos que los objetos se comparan *por referencia*: dos valores de objetos son iguales si y sólo si se refieren al mismo objeto subyacente.

```
let a = []; // La variable a se refiere a un array vacío.
let b = a; // Ahora b se refiere al mismo array. b[0] = 1; // Mutar el array al que se refiere la variable b. a[0] // => 1: el cambio también es visible a través de la variable a.
a === b // => verdadero: a y b se refieren al mismo objeto, por lo que son iguales.
```

Como puedes ver en este código, asignar un objeto (o array) a una variable simplemente asigna la referencia: no crea una nueva copia

del objeto. Si quieres hacer una nueva copia de un objeto o matriz, debes copiar explícitamente las propiedades del objeto o los elementos de la matriz. Este ejemplo demuestra el uso de un bucle `for` ([§5.4.3](#)):

```
let a = ["a", "b", "c"]; // Un array que queremos copiar let b = []; // Un array distinto en el que copiaremos for(let i = 0; i < a.length; i++) { // Por cada índice de a[] b[i] = a[i]; // Copiar un elemento de a en b } let c = Array.from(b); // En ES6, copiar arrays con Array.from()
```

Del mismo modo, si queremos comparar dos objetos o arrays distintos, debemos comparar sus propiedades o elementos. Este código define una función para comparar dos arrays:

```
function equalArrays(a, b) { if (a === b) return true; // Las matrices idénticas son iguales if (a.length !== b.length) return false; // Las matrices de tamaño diferente no son iguales for(let i = 0; i < a.length; i++) { // Recorre todos los elementos if (a[i] !== b[i]) return false; // Si alguno difiere, las matrices no son iguales } return true; // En caso contrario son iguales}
```

## 3.9 Conversiones de tipo

JavaScript es muy flexible en cuanto a los tipos de valores que requiere. Lo hemos visto en el caso de los booleanos: cuando JavaScript espera un valor booleano, puede proporcionar un valor de cualquier tipo y JavaScript lo convertirá según sea necesario. Algunos valores (valores "verdaderos") se convierten en verdaderos y otros

(valores "falsos") se convierten en falsos. Lo mismo ocurre con otros tipos: si JavaScript quiere una cadena, convertirá el valor que le des en una cadena. Si JavaScript quiere un número, intentará convertir el valor que le des en un número (o en NaN si no puede realizar una conversión significativa).

Algunos ejemplos:

**10 + " objetos" // => "10 objetos":** El número 10 se convierte en una cadena "7" \* "4"  
**// => 28:** ambas cadenas se convierten en números let n = 1 - "x"; // n == NaN; la cadena "x" no puede convertirse en un número n + " objetos" // => "NaN objetos":  
NaN se convierte en la cadena "NaN"

La Tabla 3-2 resume cómo se convierten los valores de un tipo a otro en JavaScript. Las entradas en negrita de la tabla resaltan las conversiones que pueden resultar sorprendentes. Las celdas vacías indican que no es necesaria ninguna conversión y no se realiza ninguna.

*Tabla 3-2. Conversiones de tipos de JavaScript*

| Valor                         | a la cadena  | al número | a Booleano   |
|-------------------------------|--------------|-----------|--------------|
| indefinido                    | "indefinido" | NaN       | falso        |
| null                          | "null"       | <b>0</b>  | falso        |
| verdadero                     | "verdadero"  | <b>1</b>  |              |
| falso                         | "falso"      | <b>0</b>  |              |
| "" (cadena vacía)             |              | <b>0</b>  | <b>falso</b> |
| "1,2" (no vacío, numérico)    |              | 1.2       | verdadero    |
| "uno" (no vacío, no numérico) |              | NaN       | verdadero    |
| 0                             | "0"          |           | <b>falso</b> |
| -0                            | "0"          |           | <b>falso</b> |

|                                 |                                  |  |
|---------------------------------|----------------------------------|--|
| 1 (finito, no nulo)             | "1"                              | verdadero                                |
| Infinito                        | "Infinito"                       | verdadero                                |
| -Infinity                       | "-Infinito"                      | verdadero                                |
| NaN                             | "NaN"                            | <b>falso</b>                             |
| {} (cualquier objeto)           | <i>véase el apartado 3.9.3</i>   | <i>véase el apartado 3.9.3</i> verdadero |
| [] (matriz vacía)               | ""                               | <b>0</b> verdadero                       |
| [9] (un elemento numérico)      | "9"                              | <b>9</b> verdadero                       |
| ['a'] (cualquier otra matriz)   | <i>utilizar el método join()</i> | NaN verdadero                            |
| función(){} (cualquier función) | <i>véase el apartado 3.9.3</i>   | NaN verdadero                            |

Las conversiones de primitivo a primitivo que se muestran en la tabla son relativamente sencillas. La conversión a booleano ya se discutió en §3.4.

La conversión a cadenas está bien definida para todos los valores primitivos.

La conversión a números es un poco más complicada. Las cadenas que pueden ser analizadas como números se convierten en esos números. Los espacios iniciales y finales están permitidos, pero cualquier carácter sin espacio inicial o final que no forme parte de un literal numérico hace que la conversión de cadena a número produzca NaN. Algunas conversiones numéricas pueden parecer sorprendentes: true se convierte en 1, y false y la cadena vacía se convierten en 0.

La conversión de objeto a primitivo es algo más complicada, y es el tema de §3.9.3.

### 3.9.1 Conversiones e igualdad

JavaScript tiene dos operadores que comprueban si dos valores son iguales. El "operador de igualdad estricta", `==`, no considera que sus operandos sean iguales si no son del mismo tipo, y éste es casi siempre el operador correcto a la hora de codificar. Pero como JavaScript es tan flexible con las conversiones de tipo, también define el operador `==` con una definición flexible de igualdad. Todas las siguientes comparaciones son verdaderas, por ejemplo:

**null == undefined // => true:** Estos dos valores se tratan como iguales. "0" == 0 // => true: La cadena se convierte en un número antes de comparar. 0 == false // => true: El booleano se convierte en número antes de comparar. "0" == false // => true: ¡Los dos operandos se convierten en 0 antes de comparar!

En §4.9.1 se explica exactamente qué conversiones realiza el operador `==` para determinar si dos valores deben considerarse iguales.

Tenga en cuenta que la convertibilidad de un valor a otro no implica la igualdad de esos dos valores. Si se utiliza `undefined` donde se espera un valor booleano, por ejemplo, se convertirá en `false`. Pero esto no significa que `undefined == false`. Los operadores y sentencias de JavaScript esperan valores de varios tipos y realizan conversiones a esos tipos. La sentencia `if` convierte `undefined` en `false`, pero el operador `==` nunca intenta convertir sus operandos en booleanos.

### 3.9.2 Conversiones explícitas

Aunque JavaScript realiza muchas conversiones de tipo de forma automática, a veces puede ser necesario realizar una conversión explícita, o puede que prefiera hacer las conversiones explícitas para que su código sea más claro.

La forma más sencilla de realizar una conversión de tipo explícita es utilizar las funciones Boolean(), Number() y String():

```
Number("3") //=> 3  
String(false) //=> "false": O utiliza false.toString()  
Boolean([]) //=> true
```

Cualquier valor que no sea null o undefined tiene un método `toString()`, y el resultado de este método suele ser el mismo que el devuelto por la función `String()`.

Como apunte, tenga en cuenta que las funciones Boolean(), Number() y String() también pueden ser invocadas -con new- como constructor. Si las utilizas de esta manera, obtendrás un objeto "envolvente" que se comporta como un valor booleano, numérico o de cadena primitivo. Estos objetos envolventes son un remanente histórico de los primeros días de JavaScript, y nunca hay una buena razón para utilizarlos.

Algunos operadores de JavaScript realizan conversiones de tipo implícitas y a veces se utilizan explícitamente con el fin de convertir el tipo. Si un operando del operador + es una cadena, convierte el otro en una cadena. El operador unario + convierte su operando en un número. Y el operador unario ! convierte su operando en un booleano y lo niega.

Estos hechos conducen a los siguientes modismos de conversión de tipos que puede ver en algún código:

```
x + "" //=> String(x)  
+x //=> Número(x) x-0 //=> Número(x) !! x //=>  
Boolean(x): Nota doble !
```

El formateo y el análisis sintáctico de los números son tareas habituales en los programas informáticos, y JavaScript dispone de funciones y métodos especializados que proporcionan un control más preciso sobre las conversiones de número a cadena y de cadena a número.

El método `toString()` definido por la clase `Number` acepta un argumento opcional que especifica un radix, o base, para la conversión. Si no se especifica el argumento, la conversión se realiza en base 10. Sin embargo, también se pueden convertir números en otras bases (entre 2 y 36). Por ejemplo:

```
let n = 17; let binary = "0b" + n.toString(2); // binary == "0b10001" let octal =
"0o" + n.toString(8); // octal == "0o21" let hex = "0x" + n.toString(16); // hex ==
"0x11"
```

Cuando se trabaja con datos financieros o científicos, es posible que se quiera convertir números en cadenas de manera que se pueda controlar el número de decimales o el número de dígitos significativos en la salida, o se quiera controlar si se utiliza la notación exponencial. La clase `Number` define tres métodos para este tipo de conversiones de número a cadena. `toFixed()` convierte un número en una cadena con un número especificado de dígitos después del punto decimal. Nunca utiliza la notación exponencial. `toExponential()` convierte un número en una cadena utilizando la notación exponencial, con un dígito antes del punto decimal y un número especificado de dígitos después del punto decimal (lo que significa que el número de dígitos significativos es uno mayor que el valor que se especifica). `toPrecision()` convierte un número en una cadena con el número de dígitos significativos que se especifica. Utiliza la notación exponencial si el número de dígitos significativos no es lo

suficientemente grande como para mostrar toda la parte entera del número. Tenga en cuenta que los tres métodos redondean los dígitos finales o los rellenan con ceros, según convenga. Considere los siguientes ejemplos:

```
dejemos n = 123456.789;  
n.toFixed(0) //=> "123457"  
n.toFixed(2) //=> "123456.79"  
n.toFixed(5) //=> "123456.78900"  
n.toExponential(1) //=> "1.2e+5"  
n.toExponential(3) //=> "1.235e+5"  
n.toPrecision(4) //=> "1.235e+5"  
n.toPrecision(7) //=> "123456.8"  
n.toPrecision(10) //=> "123456.7890"
```

Además de los métodos de formateo de números mostrados aquí, la clase Intl.NumberFormat define un método de formateo de números más general e internacionalizado. Véase [§11.7.1](#) para más detalles.

Si se pasa una cadena a la función de conversión Number(), ésta intenta analizar esa cadena como un entero o un literal de punto flotante. Esta función sólo funciona con números enteros de base 10 y no admite caracteres finales que no formen parte del literal. Las funciones parseInt() y

Las funciones parseFloat() (son funciones globales, no métodos de ninguna clase) son más flexibles. parseInt() analiza sólo números enteros, mientras que parseFloat() analiza tanto números enteros como de punto flotante. Si una cadena comienza con "0x" o "0X", parseInt() la interpreta como un número hexadecimal. Tanto parseInt() como parseFloat() omiten los espacios en blanco iniciales, analizan tantos caracteres numéricos como pueden e ignoran todo lo

que sigue. Si el primer carácter sin espacio no forma parte de un literal numérico válido, devuelven NaN:

```
parseInt("3 ratones ciegos") //=> 3 parseFloat(" 3,14 metros") // => 3,14 parseInt("-12,34") // => -12 parseInt("0xFF") // => 255 parseInt("0xff") // => 255 parseInt("-0xFF") // => -255 parseFloat(".1") // => 0.1 parseInt("0.1") // => 0 parseInt(".1") // => NaN: los enteros no pueden empezar por "." parseFloat("$72.47") // => NaN: los números no pueden empezar por "$"
```

parseInt() acepta un segundo argumento opcional que especifica el radix (base) del número a analizar. Los valores legales están entre 2 y 36. Por ejemplo:

```
parseInt("11", 2) //=> 3: (1*2 + 1) parseInt("ff", 16) // => 255: (15*16 + 15)  
parseInt("zz", 36) // => 1295: (35*36 + 35) parseInt("077", 8) // => 63: (7*8  
+ 7) parseInt("077", 10) // => 77: (7*10 + 7)
```

### 3.9.3 Conversiones de objetos a primitivas

Las secciones anteriores han explicado cómo se pueden convertir explícitamente valores de un tipo a otro tipo y han explicado las conversiones implícitas de JavaScript de valores de un tipo primitivo a otro tipo primitivo. Esta sección cubre las complicadas reglas que JavaScript utiliza para convertir objetos en valores primitivos. Es larga y oscura, y si ésta es su primera lectura de este capítulo, debería sentirse libre de saltar a [§3.10](#).

Una de las razones de la complejidad de las conversiones de objetos a primitivas de JavaScript es que algunos tipos de objetos tienen más de una representación primitiva. Los objetos de fecha, por ejemplo, pueden representarse como cadenas o como marcas de tiempo numéricas. La especificación de JavaScript define tres algoritmos fundamentales para convertir objetos en valores primitivos:

### *prefer-string*

Este algoritmo devuelve un valor primitivo, prefiriendo un valor de cadena, si la conversión a cadena es posible.

### *número de preferencia*

Este algoritmo devuelve un valor primitivo, prefiriendo un número, si tal conversión es posible.

### *no-preferencia*

Este algoritmo no expresa ninguna preferencia sobre el tipo de valor primitivo deseado, y las clases pueden definir sus propias conversiones. De los tipos incorporados en JavaScript, todos excepto Date implementan este algoritmo como *prefer-number*. La clase Date implementa este algoritmo como *prefer-string*.

La implementación de estos algoritmos de conversión de objetos a primitivos se explica al final de esta sección. Sin embargo, primero explicamos cómo se utilizan los algoritmos en JavaScript.

## CONVERSIONES DE OBJETO A BOOLEANO

Las conversiones de objetos a booleanos son triviales: todos los objetos se convierten en verdaderos.

Obsérvese que esta conversión no requiere el uso de los algoritmos objeto-primitivo descritos, y que se aplica literalmente a *todos los* objetos, incluidos los arrays vacíos e incluso el objeto envolvente new

Booleano(falso).

## CONVERSIONES DE OBJETO A CADENA

Cuando un objeto necesita ser convertido a una cadena, JavaScript primero lo convierte a una primitiva usando el algoritmo *prefer-*

*string*, y luego convierte el valor primitivo resultante a una cadena, si es necesario, siguiendo las reglas de la [Tabla 3-2](#).

Este tipo de conversión ocurre, por ejemplo, si se pasa un objeto a una función incorporada que espera un argumento de cadena, si se llama a

`String()` como función de conversión, y cuando se interpolan objetos en literales de plantilla ([§3.3.4](#)).

## CONVERSIONES DE OBJETOS A NÚMEROS

Cuando se necesita convertir un objeto en un número, JavaScript lo convierte primero en un valor primitivo utilizando el algoritmo *preferir-número*, y luego convierte el valor primitivo resultante en un número, si es necesario, siguiendo las reglas de la [Tabla 3-2](#).

Las funciones y los métodos de JavaScript incorporados que esperan argumentos numéricos convierten los argumentos de los objetos en números de esta manera, y la mayoría (véanse las excepciones que siguen) de los operadores de JavaScript que esperan operandos numéricos también convierten los objetos en números de esta manera.

## CONVERSIONES DE OPERADORES DE CASOS ESPECIALES

Los operadores se tratan en detalle en [el capítulo 4](#). Aquí explicamos los operadores de casos especiales que no utilizan las conversiones básicas de objeto a cadena y de objeto a número descritas anteriormente.

El operador `+` en JavaScript realiza la suma numérica y la concatenación de cadenas. Si cualquiera de sus operandos es un

objeto, JavaScript los convierte en valores primitivos utilizando el algoritmo de *no preferencia*. Una vez que tiene dos valores primitivos, comprueba sus tipos. Si alguno de los argumentos es una cadena, convierte el otro en una cadena y concatena las cadenas. En caso contrario, convierte ambos argumentos en números y los suma.

Los operadores == y != realizan pruebas de igualdad y desigualdad de una forma poco precisa que permite conversiones de tipo. Si un operando es un objeto y el otro es un valor primitivo, estos operadores convierten el objeto en primitivo utilizando el algoritmo de *no preferencia* y luego comparan los dos valores primitivos.

Por último, los operadores relacionales <, <=, > y >= comparan el orden de sus operandos y pueden utilizarse para comparar tanto números como cadenas. Si alguno de los operandos es un objeto, se convierte en un valor primitivo mediante el algoritmo de *preferencia numérica*. Tenga en cuenta, sin embargo, que a diferencia de la conversión objeto-número, los valores primitivos devueltos por la conversión preferir-número no se convierten en números.

Tenga en cuenta que la representación numérica de los objetos Date es significativamente comparable con < y >, pero la representación de cadena no lo es. Para los objetos Date, el algoritmo de *no-preferencia* convierte a una cadena, por lo que el hecho de que JavaScript utilice el algoritmo *de preferencia numérica* para estos operadores significa que podemos utilizarlos para comparar el orden de dos objetos Date.

## LOS MÉTODOS `TOSTRING()` Y `VALUEOF()`

Todos los objetos heredan dos métodos de conversión que son utilizados por las conversiones objeto-primitivo, y antes de que podamos explicar los algoritmos de conversión *preferir-cadena*, *preferir-número* y *no-preferencia*, tenemos que explicar estos dos métodos.

El primer método es `toString()`, y su trabajo es devolver una representación de cadena del objeto. El método `toString()` por defecto no devuelve un valor muy interesante (aunque lo encontraremos útil en [§14.4.3](#)):

```
{x: 1, y: 2}.toString() //=> "[objeto Objeto]"
```

Muchas clases definen versiones más específicas del método `toString()`. El método `toString()` de la clase `Array`, por ejemplo, convierte cada elemento de la matriz en una cadena y une las cadenas resultantes con comas entre ellas. El método `toString()` de la clase `Function` convierte las funciones definidas por el usuario en cadenas de código fuente de JavaScript. La clase `Date` define un método `toString()` que devuelve una cadena de fecha y hora legible para el ser humano (y que se puede analizar en JavaScript). La clase `RegExp` define un método `toString()` que convierte los objetos `RegExp` en una cadena que se parece a un literal `RegExp`:

```
[1,2,3].toString() //=> "1,2,3"  
(function(x) { f(x); }).toString() //=> "function(x { f(x); })" /\d+/g.toString() //=>  
"/\d+/g" let d = new Date(2020,0,1);  
d.toString() //=> "Wed Jan 01 2020 00:00:00 GMT-0800  
(Hora del Pacífico)"
```

La otra función de conversión de objetos se llama `valueOf()`. El trabajo de este método está menos definido: se supone que

convierte un objeto en un valor primitivo que representa el objeto, si es que existe tal valor primitivo. Los objetos son valores compuestos, y la mayoría de los objetos no pueden ser representados por un único valor primitivo, por lo que el método `valueOf()` por defecto simplemente devuelve el objeto en sí mismo en lugar de devolver una primitiva. Las clases envolventes como `String`, `Number` y `Boolean` definen métodos `valueOf()` que simplemente devuelven el valor primitivo envuelto. Las matrices, funciones y expresiones regulares simplemente heredan el método por defecto. La llamada a `valueOf()` para instancias de estos tipos simplemente devuelve el propio objeto. La clase `Date` define un método `valueOf()` que devuelve la fecha en su representación interna: el número de milisegundos desde el 1 de enero de 1970:

```
let d = new Date(2010, 0, 1); // 1 de enero de 2010, (hora del Pacífico)
d.valueOf() //=> 1262332800000
```

## ALGORITMOS DE CONVERSIÓN DE OBJETOS A PRIMITIVOS

Una vez explicados los métodos `toString()` y `valueOf()`, podemos explicar aproximadamente cómo funcionan los tres algoritmos de conversión de objetos a primitivos (los detalles completos se posponen hasta [§14.4.7](#)):

- El algoritmo *preferir-cadena* primero intenta el método `toString()`. Si el método está definido y devuelve un valor primitivo, entonces JavaScript utiliza ese valor primitivo (jaunque no sea una cadena!). Si `toString()` no existe o si devuelve un objeto, entonces JavaScript prueba el método `valueOf()`. Si ese método existe y devuelve un valor primitivo, entonces JavaScript utiliza ese valor. En caso contrario, la conversión falla con un

• `TypeError`.

- El algoritmo *preferir-número* funciona como el algoritmo *preferir-cadena*, excepto que intenta primero `valueOf()` y luego `toString()`.
- El algoritmo de *no-preferencia* depende de la clase del objeto que se está convirtiendo. Si el objeto es un objeto `Date`, JavaScript utiliza el algoritmo *prefer-string*. Para cualquier otro objeto, JavaScript utiliza el algoritmo *prefer-number*.

Las reglas descritas aquí son válidas para todos los tipos incorporados de JavaScript y son las reglas por defecto para cualquier clase que defina usted mismo. En [§14.4.7](#) se explica cómo puede definir sus propios algoritmos de conversión de objeto a primitivo para las clases que defina.

Antes de dejar este tema, vale la pena señalar que los detalles de la conversión de *número preferido* explican por qué las matrices vacías se convierten en el número 0 y las matrices de un solo elemento también pueden convertirse en números:

```
Number([]) //=> 0: esto es inesperado!
Number([99]) //=> 99: ¿en serio?
```

La conversión de objeto a número convierte primero el objeto en una primitiva utilizando el algoritmo *prefer-number*, y luego convierte el valor primitivo resultante en un número. El algoritmo *preferir-número* intenta primero `valueOf()` y luego recurre a `toString()`. Pero la clase `Array` hereda el método `valueOf()` por defecto, que no devuelve un valor primitivo. Así que cuando intentamos convertir un array en un número, acabamos invocando el método `toString()` del array. Los arrays vacíos se convierten en la

cadena vacía. Y la cadena vacía se convierte en el número 0. Un array con un solo elemento se convierte en la misma cadena que ese único elemento. Si un array contiene un solo número, ese número se convierte en una cadena, y luego de nuevo en un número.

## 3.10 Declaración y asignación de variables

Una de las técnicas más fundamentales de la programación informática es el uso de nombres -o *identificadores*- para representar valores. La vinculación de un nombre a un valor nos permite referirnos a ese valor y utilizarlo en los programas que escribimos. Cuando hacemos esto, solemos decir que estamos asignando un valor a una *variable*. El término "variable" implica que se pueden asignar nuevos valores: que el valor asociado a la variable puede variar a medida que nuestro programa se ejecuta. Si asignamos permanentemente un valor a un nombre, entonces llamamos a ese nombre una *constante* en lugar de una variable.

Antes de poder utilizar una variable o constante en un programa de JavaScript, hay que declararla. En ES6 y posteriores, esto se hace con las palabras clave `let` y `const`, que explicamos a continuación. Antes de ES6, las variables se declaraban con `var`, que es más idiosincrásico y se explica más adelante en esta sección.

### 3.10.1 Declaraciones con `let` y `const`

En el JavaScript moderno (ES6 y posteriores), las variables se declaran con la palabra clave `let`, así:

`dejar que  
i; dejar  
que sum;`

También puedes declarar múltiples variables en una sola sentencia let:

**deja que i, suma;**

Es una buena práctica de programación asignar un valor inicial a sus variables cuando las declara, cuando esto es posible:

```
let mensaje = "hola"; let i = 0, j = 0, k = 0; let x = 2, y = x*x; // Los  
inicializadores pueden utilizar variables previamente declaradas
```

Si no se especifica un valor inicial para una variable con la sentencia let, la variable se declara, pero su valor es indefinido hasta que el código le asigne un valor.

Para declarar una constante en lugar de una variable, utilice const en lugar de let. const funciona igual que let, salvo que debe inicializar la constante al declararla:

```
const H0 = 74; // Constante de Hubble (km/s/Mpc) const C = 299792.458; //  
Velocidad de la luz en el vacío (km/s) const AU = 1.496E8; // Unidad astronómica:  
distancia al sol (km)
```

Como su nombre indica, las constantes no pueden tener sus valores cambiados, y cualquier intento de hacerlo provoca un TypeError.

Es una convención común (pero no universal) declarar las constantes usando nombres con todas las letras mayúsculas como H0 o HTTP\_NOT\_FOUND como una forma de distinguirlas de las variables.

## CUÁNDO USAR CONST

Hay dos escuelas de pensamiento sobre el uso de la palabra clave const. Un enfoque es utilizar const sólo para valores que son fundamentalmente inmutables, como las constantes físicas mostradas, o los números de versión del programa, o las secuencias de bytes utilizadas para identificar los tipos de archivo, por ejemplo. Otro enfoque reconoce que muchas de las llamadas variables en nuestro programa en realidad nunca cambian mientras nuestro programa se ejecuta. En este enfoque, declaramos todo con const, y luego si encontramos que realmente queremos permitir que el valor varíe, cambiamos la declaración a let.

Esto puede ayudar a prevenir errores al descartar cambios accidentales en las variables que no pretendíamos.

En un enfoque, utilizamos const sólo para los valores que *no deben* cambiar. En el otro, usamos const para cualquier valor que no cambie. Yo prefiero el primer enfoque en mi propio código.

En [el Capítulo 5](#), aprenderemos sobre las sentencias de bucle for, for/in y for/of en JavaScript. Cada uno de estos bucles incluye una variable de bucle que recibe un nuevo valor asignado en cada iteración del bucle. JavaScript nos permite declarar la variable de bucle como parte de la propia sintaxis del bucle, y ésta es otra forma común de utilizar let:

```
for(let i = 0, len = data.length; i < len; i++) console.log(data[i]); for(let  
datum of data) console.log(datum); for(let property in object)  
console.log(property);
```

Puede parecer sorprendente, pero también se puede utilizar const para declarar las "variables" del bucle para los bucles for/in y for/of, siempre que el cuerpo del bucle no reasigne un nuevo valor. En este caso, la declaración const sólo está diciendo que el valor es constante durante la duración de una iteración del bucle:

```
for(const datum of data) console.log(datum);
```

```
for(const propiedad en objeto) console.log(propiedad);
```

## ÁMBITO VARIABLE Y CONSTANTE

El *ámbito* de una variable es la región del código fuente del programa en la que está definida. Las variables y constantes declaradas con let y const tienen un *ámbito de bloque*. Esto significa que sólo se definen dentro del bloque de código en el que aparece la sentencia let o const. Las definiciones de clases y funciones de JavaScript son bloques, al igual que los cuerpos de las sentencias if/else, los bucles while, los bucles for, etc. A grandes rasgos, si una variable o una constante se declara dentro de un conjunto de llaves, esas llaves delimitan la región de código en la que se define la variable o la constante (aunque, por supuesto, no es legal hacer referencia a una variable o una constante desde las líneas de código que se ejecutan antes de la sentencia let o const que declara la variable). Las variables y constantes declaradas como parte de un bucle for, for/in, o for/of tienen como *ámbito* el cuerpo del bucle, aunque técnicamente aparezcan fuera de las llaves.

Cuando una declaración aparece en el nivel superior, fuera de cualquier bloque de código, decimos que es una variable o constante *global* y que tiene alcance global. En Node y en los módulos JavaScript del lado del cliente (véase [el capítulo 10](#)), el *ámbito* de una variable global es el archivo en el que está definida. Sin embargo, en el JavaScript tradicional del lado del cliente, el *ámbito* de una variable global es el documento HTML en el que está definida. Es decir: si un <script> declara una variable o constante global, esa variable o constante se define en todos los elementos <script> de ese

documento (o al menos en todos los scripts que se ejecutan después de la sentencia let o const).

## DECLARACIONES REPETIDAS

Es un error sintáctico utilizar el mismo nombre con más de una declaración let o const en el mismo ámbito. Es legal (aunque es una práctica que es mejor evitar) declarar una nueva variable con el mismo nombre en un ámbito anidado:

```
const x = 1; // Declara x como una constante global if (x === 1) { let x = 2; //
Dentro de un bloque x puede referirse a un valor diferente console.log(x); //
Imprime 2 } console.log(x); // Imprime 1: ahora estamos de vuelta en el ámbito
global let x = 3; // ¡Error! Error de sintaxis al intentar redeclarar x
```

## DECLARACIONES Y TIPOS

Si estás acostumbrado a lenguajes de tipado estático como C o Java, puedes pensar que el propósito principal de las declaraciones de variables es especificar el tipo de valores que pueden ser asignados a una variable. Pero, como has visto, no hay ningún tipo asociado a las declaraciones de variables de JavaScript.<sup>2 Una variable de JavaScript</sup> puede contener un valor de cualquier tipo. Por ejemplo, es perfectamente legal (pero generalmente de mal estilo de programación) en JavaScript asignar un número a una variable y luego asignar una cadena a esa variable:

```
deje i = 10; i =
"diez";
```

### 3.10.2 Declaraciones de variables con var

En las versiones de JavaScript anteriores a ES6, la única forma de declarar una variable es con la palabra clave var, y no hay forma de declarar constantes. La sintaxis de var es igual que la de let:

```
var x; var data = [], count = data.length;  
for(var i = 0; i < count; i++) console.log(data[i]);
```

Aunque var y let tienen la misma sintaxis, existen importantes diferencias en su funcionamiento:

- Las variables declaradas con var no tienen ámbito de bloque. En su lugar, tienen un ámbito de aplicación en el cuerpo de la función que las contiene, independientemente de la profundidad con la que estén anidadas dentro de esa función.
- Si se utiliza var fuera del cuerpo de una función, se declara una variable global. Pero las variables globales declaradas con var difieren de las globales declaradas con let en un aspecto importante. Las globales declaradas con var se implementan como propiedades del objeto global ([§3.7](#)). El objeto global puede ser referenciado como globalThis. Así que si escribes var x = 2; fuera de una función, es como si hubieras escrito globalThis.x = 2;. Tenga en cuenta, sin embargo, que la analogía no es perfecta: las propiedades creadas con declaraciones var globales no pueden borrarse con el operador delete ([§4.13.4](#)). Las variables y constantes globales declaradas con let y const no son propiedades del objeto global.
- A diferencia de las variables declaradas con let, es legal declarar la misma variable varias veces con var. Y como las variables var tienen ámbito de función en lugar de ámbito de bloque, es realmente común hacer este tipo de redeclaración. La variable i se utiliza frecuentemente para valores enteros, y especialmente como variable índice de los

bucles for. En una función con múltiples bucles for, es típico que cada uno comience `for(var i = 0; ....`. Debido a que var no abarca estas variables en el cuerpo del bucle, cada uno de estos bucles está (inofensivamente) re-declarando y reiniciando la misma variable.

- Una de las características más inusuales de las declaraciones `var` se conoce como *elevación*. Cuando se declara una variable con `var`, la declaración se eleva (o "eleva") a la parte superior de la función que la encierra. La inicialización de la variable permanece donde la escribiste, pero la definición de la variable se mueve a la parte superior de la función. Por lo tanto, las variables declaradas con `var` pueden utilizarse, sin error, en cualquier parte de la función adjunta. Si el código de inicialización no se ha ejecutado todavía, entonces el valor de la variable puede ser indefinido, pero no obtendrá un error si utiliza la variable antes de que sea inicializada. (Esto puede ser una fuente de errores y es uno de los errores importantes que `let` corrige: si declaras una variable con `let` pero intentas usarla antes de que se ejecute la sentencia `let`, obtendrás un error real en lugar de ver simplemente un valor indefinido).

## UTILIZANDO VARIABLES NO DECLARADAS

En modo estricto ([§5.6.3](#)), si intentas usar una variable no declarada, obtendrás un error de referencia cuando ejecutes tu código. Sin embargo, fuera del modo estricto, si asignas un valor a un nombre que no ha sido declarado con `let`, `const` o `var`, acabarás creando una nueva variable global. Será una global sin importar la profundidad de anidación de tu código dentro de las funciones y bloques, lo que casi con seguridad no es lo que quieras, es propenso a errores y es una de las mejores razones para usar el modo estricto.

Las variables globales creadas de esta manera accidental son como las variables globales declaradas con `var`: definen propiedades del objeto global. Pero a diferencia de las propiedades definidas por declaraciones `var`

•  
adecuadas, estas propiedades *pueden* ser eliminadas con el operador delete  
[\(§4.13.4\)](#).

### 3.10.3 Asignación de desestructuración

ES6 implementa un tipo de declaración compuesta y sintaxis de asignación conocida como **asignación de desestructuración**. En una asignación de desestructuración, el valor del lado derecho del signo igual es un array o un objeto (un valor "estrukturado"), y el lado izquierdo especifica uno o más nombres de variables utilizando una sintaxis que imita la sintaxis literal de los arrays y los objetos.

Cuando se produce una asignación de desestructuración, se extraen uno o más valores ("desestructurados") del valor de la derecha y se almacenan en las variables nombradas en la izquierda. La asignación de desestructuración es quizás la más utilizada para inicializar variables como parte de una declaración const, let o var, pero también puede hacerse en expresiones de asignación regulares (con variables que ya han sido declaradas). Y, como veremos en [§8.3.5](#), la **desestructuración** también puede utilizarse al definir los parámetros de una función.

A continuación se presentan asignaciones sencillas de desestructuración utilizando matrices de valores:

```
let [x,y] = [1,2]; // Igual que let x=1, y=2 [x,y] = [x+1,y+1]; // Igual que x =  
x + 1, y = y + 1  
[x,y] = [y,x]; // Intercambiar el valor de las dos variables  
[x,y] // => [3,2]: los valores incrementados e intercambiados
```

Observe cómo la asignación de desestructuración facilita el trabajo con funciones que devuelven matrices de valores:

```
// Convierte las coordenadas [x,y] en coordenadas polares [r,theta] function  
toPolar(x, y) { return [Math.sqrt(x*x+y*y), Math.atan2(y,x)]; }  
  
// Convertir coordenadas polares en cartesianas
```

```
function toCartesian(r, theta) { return [r*Math.cos(theta), r*Math.sin(theta)]; }

let [r,theta] = toPolar(1.0, 1.0); // r == Math.sqrt(2); theta == Math.PI/4 let [x,y] = toCartesian(r,theta); // [x, y] == [1.0, 1,0]
```

Hemos visto que las variables y las constantes pueden declararse como parte de los distintos bucles for de JavaScript. También es posible utilizar la desestructuración de variables en este contexto. Aquí hay un código que hace un bucle sobre los pares nombre/valor de todas las propiedades de un objeto y utiliza la asignación de desestructuración para convertir esos pares de matrices de dos elementos en variables individuales:

```
let o = { x: 1, y: 2 }; // El objeto sobre el que haremos un bucle for(const [nombre, valor] of Object.entries(o)) { console.log(nombre, valor); // Imprime "x 1" e "y 2" }
```

El número de variables a la izquierda de una asignación de desestructuración no tiene por qué coincidir con el número de elementos del array a la derecha. Las variables extra a la izquierda se establecen como indefinidas, y los valores extra a la derecha se ignoran. La lista de variables de la izquierda puede incluir comas adicionales para omitir ciertos valores de la derecha:

```
let [x,y] = [1]; // x == 1; y == indefinido [x,y] = [1,2,3]; // x == 1; y == 2 [x,,y] = [1,2,3,4]; // x == 2; y == 4
```

Si desea reunir todos los valores no utilizados o restantes en una sola variable al desestructurar una matriz, utilice tres puntos (...) antes del último nombre de la variable en el lado izquierdo:

```
let [x, ... y] = [1,2,3,4]; // y == [2,3,4]
```

Volveremos a ver tres puntos utilizados de esta manera en §8.3.2, donde se utilizan para indicar que todos los argumentos restantes de la función deben recogerse en una única matriz.

La asignación desestructurada puede utilizarse con arrays anidados. En este caso, el lado izquierdo de la asignación debe parecerse a un literal de matriz anidada:

```
let [a, [b, c]] = [1, [2,2.5], 3]; // a == 1; b == 2; c == 2.5
```

Una poderosa característica de la desestructuración de arrays es que, en realidad, no requiere un array. Se puede utilizar cualquier objeto *iterable* ([Capítulo 12](#)) en el lado derecho de la asignación; también se puede desestructurar cualquier objeto que se pueda utilizar con un bucle for/of ([§5.4.4](#)):

```
let [first, ... rest] = "Hola"; // first == "H"; rest == ["o", "l", "l", "o"]
```

La asignación de desestructuración también puede realizarse cuando el lado derecho es un valor de objeto. En este caso, el lado izquierdo de la asignación se parece a un literal de objeto: una lista separada por comas de nombres de variables entre llaves:

```
let transparent = {r: 0.0, g: 0.0, b: 0.0, a: 1.0}; // Un color RGBA  
let {r, g, b} = transparent; // r == 0.0; g == 0.0; b == 0.0
```

El siguiente ejemplo copia las funciones globales del objeto Math en variables, lo que podría simplificar el código que hace mucha trigonometría:

```
// Igual que const sin=Math.sin, cos=Math.cos, tan=Math.tan  
const {sin, cos, tan} = Math;
```

Observe en el código que el objeto Math tiene muchas propiedades además de las tres que se desestructuran en variables individuales. Las que no tienen nombre son simplemente ignoradas. Si el lado izquierdo de esta asignación hubiera incluido una variable cuyo nombre no fuera una propiedad de Math, esa variable sería simplemente asignada como indefinida.

En cada uno de estos ejemplos de desestructuración de objetos, hemos elegido nombres de variables que coinciden con los nombres de las propiedades del objeto que estamos desestructurando. Esto mantiene la sintaxis simple y fácil de entender, pero no es necesario. Cada uno de los identificadores del lado izquierdo de una asignación de desestructuración de objetos puede ser también un par de identificadores separados por dos puntos, donde el primero es el nombre de la propiedad cuyo valor se va a asignar y el segundo es el nombre de la variable a la que se va a asignar:

```
// Igual que const coseno = Math.cos, tangente = Math.tan; const { cos: coseno, tan: tangente } = Math;
```

Encuentro que la sintaxis de desestructuración de objetos se vuelve demasiado complicada para ser útil cuando los nombres de las variables y los nombres de las propiedades no son los mismos, y tiendo a evitar la taquigrafía en este caso. Si decide utilizarla, recuerde que los nombres de las propiedades están siempre a la izquierda de los dos puntos, tanto en los literales de los objetos como en la izquierda de una asignación de desestructuración de objetos.

La asignación de desestructuración se complica aún más cuando se utiliza con objetos anidados, o matrices de objetos, u objetos de matrices, pero es legal:

```
let points = [{x: 1, y: 2}, {x: 3, y: 4}]; // Un array de dos objetos punto let [{x: x1, y: y1}, {x: x2, y: y2}] = points; // desestructurado en 4 variables. (x1 === 1 && y1 === 2 && x2 === 3 && y2 === 4) // => true
```

O, en lugar de desestructurar un array de objetos, podríamos desestructurar un objeto de arrays:

```
let points = { p1: [1,2], p2: [3,4] }; // Un objeto con 2 array props let { p1: [x1, y1], p2: [x2, y2] } = puntos; // desestructurado en 4 vars (x1 === 1 && y1 === 2 && x2 === 3 && y2 === 4) // => true
```

## ENTENDER LA DESESTRUCTURACIÓN COMPLEJA

Si te encuentras trabajando con código que utiliza asignaciones de desestructuración complejas, hay una regularidad útil que puede ayudarte a dar sentido a los casos complejos. Piense primero en una asignación regular (de un solo valor). Una vez realizada la asignación, puedes tomar el nombre de la variable del lado izquierdo de la asignación y utilizarlo como una expresión en tu código, donde se evaluará al valor que le hayas asignado. Lo mismo ocurre con la asignación de desestructuración. El lado izquierdo de una asignación de desestructuración se parece a un literal de matriz o a un literal de objeto (§6.2.1 y §6.10). Después de la

el lado izquierdo funcionará como un literal de matriz o un literal de objeto válido en otra parte del código. Puede comprobar que ha escrito una asignación de desestructuración correctamente intentando utilizar el lado izquierdo en el lado derecho de otra expresión de asignación:

```
// Empezar con una estructura de datos y una desestructuración compleja let points =  
[{x: 1, y: 2}, {x: 3, y: 4}]; let [{x: x1, y: y1}, {x: x2, y: y2}] = points;  
  
// Comprueba tu sintaxis de desestructuración invirtiendo la asignación let points2 = [{x: x1, y: y1}, {x: x2, y: y2}]; // points2 == points
```

Una sintaxis de desestructuración compleja como ésta puede ser difícil de escribir y de leer, y puede ser mejor que escribas tus asignaciones explícitamente con código tradicional como let x1 =

```
points.p1[0];
```





## .11 Resumen

Algunos puntos clave que hay que recordar sobre este capítulo:

- Cómo escribir y manipular números y cadenas de texto en JavaScript.
- Cómo trabajar con los otros tipos primitivos de JavaScript: booleanos, símbolos, nulos e indefinidos.
- Las diferencias entre los tipos primitivos inmutables y los tipos de referencia mutables.
- Cómo JavaScript convierte valores implícitamente de un tipo a otro y cómo puede hacerlo explícitamente en sus programas.
- Cómo declarar e inicializar constantes y variables (incluso con asignación de desestructuración) y el ámbito léxico de las variables y constantes que se declaran.

---

Este es el formato para los números de tipo double en Java, C++ y la mayoría de los lenguajes de programación <sup>1</sup>modernos.

Hay extensiones de JavaScript, como TypeScript y Flow ([§17.8](#)), que permiten especificar tipos como parte de las declaraciones de variables con una sintaxis como let x: number = 0;

## Capítulo 4. Expresiones y operadores

---

Este capítulo documenta las expresiones de JavaScript y los operadores con los que se construyen muchas de esas expresiones.

Una *expresión* es una frase de JavaScript que puede ser *evaluada* para producir un valor. Una constante incrustada literalmente en tu programa es un tipo de expresión muy simple. Un nombre de variable es también una expresión simple que se evalúa a cualquier valor que se haya asignado a esa variable. Las expresiones complejas se construyen a partir de expresiones más simples. Una expresión de acceso a un array, por ejemplo, consiste en una expresión que se evalúa a un array seguida de un corchete abierto, una expresión que se evalúa a un entero y un corchete cerrado. Esta nueva expresión, más compleja, se evalúa al valor almacenado en el índice especificado de la matriz especificada. Del mismo modo, una expresión de invocación a una función consiste en una expresión que se evalúa a un objeto de la función y cero o más expresiones adicionales que se utilizan como argumentos de la función.

La forma más común de construir una expresión compleja a partir de expresiones más simples es con un *operador*. Un operador combina los valores de sus operandos (normalmente dos de ellos) de alguna manera y se evalúa a un nuevo valor. El operador de multiplicación \* es un ejemplo sencillo. La expresión  $x * y$  se evalúa como el producto de los valores de las expresiones  $x$  e  $y$ . Para simplificar, a veces decimos que un operador *devuelve* un valor en lugar de "evaluarse en" un valor.

Este capítulo documenta todos los operadores de JavaScript, y también explica las expresiones (como la indexación de matrices y la invocación de funciones) que no utilizan operadores. Si ya conoces otro lenguaje de programación que utilice la sintaxis del estilo C, verás que la sintaxis de la mayoría de las expresiones y operadores de JavaScript ya te resulta familiar.

## 4.1 Expresiones primarias

Las expresiones más simples, conocidas como *expresiones primarias*, son aquellas que se mantienen solas, no incluyen ninguna expresión más simple. Las expresiones primarias en JavaScript son valores constantes o *literales*, ciertas palabras clave del lenguaje y referencias a variables.

Los literales son valores constantes que se incrustan directamente en su programa. Tienen el siguiente aspecto:

```
1.23 // Un literal de número "hola" // Un literal de  
cadena  
/patrón/ // Una expresión regular literal
```

La sintaxis de JavaScript para los literales de números se trató en §3.2. Los literales de cadena se documentaron en §3.3. La sintaxis de los literales de expresiones regulares se introdujo en §3.3.5 y se documentará en detalle en §11.3.

Algunas de las palabras reservadas de JavaScript son expresiones primarias:

```
true // Evalúa el valor booleano true false // Evalúa el valor booleano  
false null // Evalúa el valor null this // Evalúa el objeto "actual"
```

Hemos aprendido sobre true, false y null en §3.4 y §3.5. A diferencia de las otras palabras clave, this no es una constante - se evalúa a diferentes valores en diferentes lugares del programa. La palabra clave this se utiliza en la programación orientada a objetos. Dentro del cuerpo de un método, this se evalúa como el objeto sobre el que se invoca el método. Véase §4.5, el capítulo 8 (especialmente §8.2.2) y el capítulo 9 para más información.

Finalmente, el tercer tipo de expresión primaria es una referencia a una variable, constante o propiedad del objeto global:

```
i // Se evalúa al valor de la variable i. sum // Se evalúa al valor de la variable sum.  
undefined // El valor de la propiedad "undefined" del objeto global
```

Cuando cualquier identificador aparece por sí mismo en un programa, JavaScript asume que es una variable o constante o propiedad del objeto global y busca su valor. Si no existe ninguna variable con ese nombre, el intento de evaluar una variable inexistente arroja un ReferenceError en su lugar.

## 4.2 Inicializadores de objetos y matrices

Los inicializadores *de objetos* y *arrays* son expresiones cuyo valor es un objeto o un array recién creado. Estas expresiones inicializadoras se denominan a veces literales de *objeto* y *literales de matriz*. Sin embargo, a diferencia de los literales verdaderos, no son expresiones primarias, porque incluyen una serie de subexpresiones que especifican valores de propiedades y elementos. Los inicializadores de arrays tienen una sintaxis un poco más simple, y comenzaremos con ellos.

Un inicializador de matriz es una lista separada por comas de expresiones contenidas entre corchetes. El valor de un inicializador de matriz es una matriz recién creada. Los elementos de esta nueva matriz se inicializan con los valores de las expresiones separadas por comas:

```
[] // Una matriz vacía: ninguna expresión dentro de los paréntesis significa que no  
// hay elementos [1+2,3+4] // Una matriz de 2 elementos. El primer elemento es 3, el  
// segundo es 7
```

Las expresiones de elementos en un inicializador de array pueden ser a su vez inicializadores de array, lo que significa que estas expresiones pueden crear arrays anidados:

```
let matrix = [[1,2,3], [4,5,6], [7,8,9]];
```

Las expresiones de elementos en un inicializador de matriz se evalúan cada vez que se evalúa el inicializador de matriz. Esto significa que el valor de una expresión del inicializador de matriz puede ser diferente cada vez que se evalúa.

Los elementos no definidos pueden incluirse en un literal de matriz simplemente omitiendo un valor entre comas. Por ejemplo, el siguiente array contiene cinco elementos, incluyendo tres elementos indefinidos:

```
let sparseArray = [1,,,5];
```

Se permite una sola coma después de la última expresión en un inicializador de matriz y no crea un elemento indefinido. Sin embargo, cualquier expresión de acceso al array para un índice posterior al de la última expresión se evaluará necesariamente como indefinido.

Las expresiones de inicialización de objetos son como las expresiones de inicialización de arrays, pero los corchetes se sustituyen por llaves, y cada subexpresión va precedida de un nombre de propiedad y dos puntos:

```
let p = { x: 2.3, y: -1.2 }; // Un objeto con 2 propiedades let q = {};// Un objeto vacío sin propiedades  
q.x = 2.3; q.y = -1.2; // Ahora q tiene las mismas propiedades que p
```

En ES6, los literales de objeto tienen una sintaxis mucho más rica en características (puede encontrar detalles en [§6.10](#)). Los literales de objeto pueden ser anidados. Por ejemplo:

```
let rectangle = { upperLeft: { x: 2, y: 2 },
  lowerRight: { x: 4, y: 5 }};
```

Volveremos a ver los inicializadores de objetos y arrays en los capítulos [6](#) y [7](#).

## 4.3 Expresiones de definición de funciones

Una expresión de definición de *función* define una función de JavaScript, y el valor de dicha expresión es la función recién definida. En cierto sentido, una expresión de definición de función es un "literal de función" del mismo modo que un inicializador de objeto es un "literal de objeto". Una expresión de definición de función suele consistir en la palabra clave `function` seguida de una lista separada por comas de cero o más identificadores (los nombres de los parámetros) entre paréntesis y un bloque de código JavaScript (el cuerpo de la función) entre llaves. Por ejemplo:

```
// Esta función devuelve el cuadrado del valor que se le pasa. let square =
function(x) { return x * x;};
```

Una expresión de definición de función también puede incluir un nombre para la función. Las funciones también pueden definirse utilizando una declaración de función en lugar de una expresión de función. Y en ES6 y posteriores, las expresiones de función pueden utilizar una nueva y compacta sintaxis de "función de flecha". Los detalles completos sobre la definición de funciones se encuentran en el [capítulo 8](#).

## 4.4 Expresiones de acceso a la propiedad

Una *expresión de acceso a una propiedad* se evalúa al valor de una propiedad de un objeto o de un elemento de un array. JavaScript define dos sintaxis para el acceso a propiedades:

```
expresión . identificador expresión [ expresión  
 ]
```

El primer estilo de acceso a la propiedad es una expresión seguida de un punto y un identificador. La expresión especifica el objeto, y el identificador especifica el nombre de la propiedad deseada. El segundo estilo de acceso a la propiedad sigue a la primera expresión (el objeto o la matriz) con otra expresión entre corchetes. Esta segunda expresión especifica el nombre de la propiedad deseada o el índice del elemento del array deseado.

He aquí algunos ejemplos concretos:

```
let o = {x: 1, y: {z: 3}}; // Un objeto de ejemplo let a = [o, 4, [5, 6]]; // Un array de  
ejemplo que contiene el objeto  
o.x //=> 1: propiedad x de la expresión o  
o.y.z //=> 3: propiedad z de la expresión  
o.y.o["x"] //=> 1: propiedad x del objeto o a[1] // => 4: elemento en el índice 1 de la  
expresión a a[2]["1"] // => 6: elemento en el índice 1 de la expresión a[2]  
a[0].x //=> 1: propiedad x de la expresión a[0]
```

Con cualquiera de los dos tipos de expresiones de acceso a propiedades, se evalúa primero la expresión que precede a . o [. Si el valor es nulo o indefinido, la expresión lanza un `TypeError`, ya que estos son los dos valores de JavaScript que no pueden tener propiedades. Si la expresión del objeto va seguida de un punto y un identificador, se busca el valor de la propiedad nombrada por ese identificador y se convierte en el valor global de la expresión. Si la expresión del objeto va seguida de otra expresión entre corchetes,

esa segunda expresión se evalúa y se convierte en una cadena. El valor global de la expresión es entonces el valor de la propiedad nombrada por esa cadena. En cualquier caso, si la propiedad nombrada no existe, el valor de la expresión de acceso a la propiedad es indefinido.

La sintaxis *.identifier* es la más sencilla de las dos opciones de acceso a propiedades, pero ten en cuenta que sólo puede utilizarse cuando la propiedad a la que quieras acceder tiene un nombre que es un identificador legal, y cuando conoces el nombre cuando escribes el programa. Si el nombre de la propiedad incluye espacios o caracteres de puntuación, o cuando es un número (para matrices), debe utilizar la notación de corchetes. Los corchetes también se utilizan cuando el nombre de la propiedad no es estático, sino que es el resultado de un cálculo (véase un ejemplo en [§6.3.1](#)).

Los objetos y sus propiedades se tratan con detalle en [el capítulo 6](#), y las matrices y sus elementos en [el capítulo 7](#).

#### 4.4.1 Acceso condicional a la propiedad

ES2020 añade dos nuevos tipos de expresiones de acceso a propiedades:

```
expresión ?. identifier expresión ?.[  
expresión ]
```

En JavaScript, los valores null e undefined son los únicos dos valores que no tienen propiedades. En una expresión regular de acceso a propiedades que utilice . o [], se obtiene un TypeError si la expresión de la izquierda se evalúa como null o undefined. Puede utilizar la sintaxis ?. y ?[] para protegerse de este tipo de errores.

Considere la expresión `a?.b`. Si `a` es nulo o indefinido, entonces la expresión se evalúa como indefinido sin ningún intento de acceder a la propiedad `b`. Si `a` es algún otro valor, entonces `a?.b` se evalúa como `a.b` (y si `a` no tiene una propiedad llamada `b`, entonces el valor será de nuevo indefinido).

Esta forma de expresión de acceso a la propiedad se denomina a veces "encadenamiento opcional" porque también funciona para expresiones de acceso a la propiedad "encadenadas" más largas como ésta:

```
dejar a = { b: null };
a. b?. c. d //=> indefinido
```

`a` es un objeto, por lo que `a.b` es una expresión de acceso a la propiedad válida. Pero el valor de `a.b` es nulo, por lo que `a.b.c` lanzaría un `TypeError`. Usando `?` en lugar de `.` evitamos el `TypeError`, y `a.b?.c` se evalúa como indefinido. Esto significa que `(a.b?.c).d` lanzará un `TypeError`, porque esa expresión intenta acceder a una propiedad de valor indefinido. Pero -y esto es una parte muy importante del "encadenamiento opcional"- `a.b?.c.d` (sin los paréntesis) simplemente se evalúa como indefinido y no arroja un error. Esto se debe a que el acceso a la propiedad con `?` es un "cortocircuito": si la subexpresión a la izquierda de `?` se evalúa como nula o indefinida, entonces toda la expresión se evalúa inmediatamente como indefinida sin ningún otro intento de acceso a la propiedad.

Por supuesto, si `a.b` es un objeto, y si ese objeto no tiene una propiedad llamada `c`, entonces `a.b?.c.d` volverá a lanzar un `TypeError`, y querremos usar otro acceso condicional a la propiedad:

```
dejar a = { b: {} };
a. b?. c?. d // => indefinido
```

El acceso condicional a las propiedades también es posible utilizando `?.[ ]` en lugar de `[ ]`. En la expresión `a?.[b][c]`, si el valor de `a` es nulo o indefinido, toda la expresión se evalúa inmediatamente como indefinida, y las subexpresiones `b` y `c` ni siquiera se evalúan. Si cualquiera de esas expresiones tiene efectos secundarios, el efecto secundario no se producirá si `a` no está definido:

```
let a; // ¡Uy, nos olvidamos de inicializar esta variable!
let index = 0; try { a[index++]; // Throws TypeError
} catch(e) { index // => 1: el incremento se produce antes de que se lance el TypeError }
a?.[index++] // => undefined: porque a es undefined index // => 1: no se incrementa
porque ?.[] cortocircuita a[index++] // !TypeError: no se puede indexar undefined.
```

El acceso condicional a propiedades con `?.` y `?.[ ]` es una de las características más nuevas de JavaScript. Desde principios de 2020, esta nueva sintaxis es compatible con las versiones actuales o beta de la mayoría de los principales navegadores.

## 4.5 Expresiones de invocación

Una expresión de *invocación* es la sintaxis de JavaScript para llamar (o ejecutar) una función o método. Comienza con una expresión de función que identifica la función a llamar. La expresión de la función va seguida de un paréntesis de apertura, una lista separada por comas de cero o más expresiones de argumentos y un paréntesis de cierre. Algunos ejemplos:

`f(0) // f es la expresión de la función; 0 es la expresión del argumento. Math.`

`max(x,y,z) // Math.max es la función; x, y y z son los argumentos.`

`a.sort() // a.sort es la función; no hay argumentos.`

Cuando se evalúa una expresión de invocación, primero se evalúa la expresión de la función y luego se evalúan las expresiones de los argumentos para producir una lista de valores de los argumentos. Si el valor de la expresión de la función no es una función, se lanza un `TypeError`. A continuación, los valores de los argumentos se asignan, en orden, a los nombres de los parámetros especificados cuando se definió la función, y luego se ejecuta el cuerpo de la función. Si la función utiliza una sentencia `return` para devolver un valor, ese valor se convierte en el valor de la expresión de invocación. En caso contrario, el valor de la expresión de invocación es indefinido. Los detalles completos sobre la invocación de funciones, incluyendo una explicación de lo que ocurre cuando el número de expresiones de argumentos no coincide con el número de parámetros en la definición de la función, se encuentran en [el Capítulo 8](#).

Toda expresión de invocación incluye un par de paréntesis y una expresión antes del paréntesis abierto. Si esa expresión es una expresión de acceso a una propiedad, entonces la invocación se conoce como una *invocación a un método*. En las invocaciones a métodos, el objeto o array objeto del acceso a propiedades se convierte en el valor de la palabra clave `this` mientras se ejecuta el cuerpo de la función. Esto permite un paradigma de programación orientado a objetos en el que las funciones (que llamamos "métodos" cuando se usan de esta manera) operan sobre el objeto del que forman parte. Véase [el capítulo 9](#) para más detalles.

#### 4.5.1 Invocación condicional

En ES2020, también se puede invocar una función utilizando `?()` en lugar de `()`. Normalmente, cuando se invoca una función, si la expresión a la izquierda de los paréntesis es nula o indefinida o cualquier otra no-función, se lanza un `TypeError`. Con la nueva sintaxis de invocación `?()`, si la expresión a la izquierda del paréntesis es nula o indefinida, toda la expresión de la invocación se evalúa como indefinida y no se lanza ninguna excepción.

Los objetos array tienen un método `sort()` al que se le puede pasar opcionalmente un argumento de función que defina el orden de clasificación deseado para los elementos del array. Antes de ES2020, si querías escribir un método como `sort()` que tomaba un argumento de función opcional, normalmente utilizabas una sentencia `if` para comprobar que el argumento de la función estaba definido antes de invocarlo en el cuerpo del `if`:

```
function square(x, log) { // El segundo argumento es un
  función opcional if (log) { // Si se pasa la función opcional log(x); // Invócala }
  return x * x; // Devuelve el cuadrado del argumento }
```

Sin embargo, con esta sintaxis de invocación condicional de ES2020, puedes simplemente escribir la invocación de la función utilizando `?()`, sabiendo que la invocación sólo se producirá si realmente hay un valor que invocar:

```
function square(x, log) { // El segundo argumento es una función opcional log?(x);
  // Llama a la función si la hay return x * x; // Devuelve el cuadrado del argumento
}
```

Tenga en cuenta, sin embargo, que `?()` sólo comprueba si el lado izquierdo es nulo o indefinido. No verifica que el valor sea realmente

una función. Así que la función square() de este ejemplo seguiría lanzando una excepción si se le pasaran dos números, por ejemplo.

Al igual que las expresiones de acceso a propiedades condicionales ([§4.4.1](#)), la invocación de funciones con ?.() es un cortocircuito: si el valor a la izquierda de ?. es nulo o indefinido, no se evalúa ninguna de las expresiones de argumentos dentro del paréntesis:

```
let f = null, x = 0; try { f(x++); // Lanza TypeError porque f es null
} catch(e) {
  x // => 1: x se incrementa antes de que se lance la excepción } f?(x++) // =>
indefinido: f es nulo, pero no se lanza la excepción x // => 1: se salta el incremento
debido al cortocircuito
```

Las expresiones de invocación condicional con ?.() funcionan tan bien para los métodos como para las funciones. Pero como la invocación de métodos también implica el acceso a propiedades, vale la pena tomarse un momento para asegurarse de entender las diferencias entre las siguientes expresiones:

**o. m()** // Acceso regular a propiedades, *invocación regular*  
**o?. m()** // Acceso condicional a propiedades, *invocación regular*  
**o. m?.()** // Acceso regular a propiedades, *invocación condicional*

En la primera expresión, o debe ser un objeto con una propiedad m y el valor de esa propiedad debe ser una función. En la segunda expresión, si o es nulo o indefinido, entonces la expresión se evalúa como indefinida. Pero si o tiene cualquier otro valor, entonces debe tener una propiedad m cuyo valor sea una función. Y en la tercera expresión, o



























no debe ser nulo o indefinido. Si no tiene una propiedad m, o si el valor de esa propiedad es nulo, entonces toda la expresión se evalúa como indefinida.

La invocación condicional con `?()` es una de las características más nuevas de JavaScript. A partir de los primeros meses de 2020, esta nueva sintaxis es compatible con las versiones actuales o beta de la mayoría de los principales navegadores.

## 4.6 Expresiones de creación de objetos

Una expresión de creación de *objeto* crea un nuevo objeto e invoca una función (llamada constructor) para inicializar las propiedades de ese objeto. Las expresiones de creación de objetos son como las expresiones de invocación, salvo que van precedidas de la palabra clave `new`:

```
new Object() new  
Point(2,3)
```

Si no se pasan argumentos a la función constructora en una expresión de creación de objetos, se puede omitir el par de paréntesis vacío:

```
nuevo Objeto  
nueva Fecha
```

El valor de una expresión de creación de objeto es el objeto recién creado.

Los constructores se explican con más detalle en [el capítulo 9](#).

## 4.7 Visión general del operador

Los operadores se utilizan para las expresiones aritméticas de JavaScript, las expresiones de comparación, las expresiones lógicas, las expresiones de asignación, etc. [La Tabla 4-1](#) resume los operadores y sirve como una conveniente referencia.

Tenga en cuenta que la mayoría de los operadores están representados por caracteres de puntuación como + y =. Algunos, sin embargo, están representados por palabras clave como delete e instanceof. Los operadores con palabras clave son operadores regulares, al igual que los expresados con signos de puntuación; simplemente tienen una sintaxis menos sucinta.

[La tabla 4-1](#) está organizada por la precedencia de los operadores. Los operadores listados en primer lugar tienen mayor precedencia que los listados en último lugar. Los operadores separados por una línea horizontal tienen diferentes niveles de precedencia. La columna A indica la asociatividad del operador, que puede ser L (izquierda-derecha) o R (derecha-izquierda), y la columna N especifica el número de operandos. La columna denominada Tipos indica los tipos esperados de los operandos y (después del símbolo →) el tipo de resultado del operador. Las subsecciones que siguen a la tabla explican los conceptos de precedencia, asociatividad y tipo de operando. Los operadores mismos se documentan individualmente después de esa discusión.

*Tabla 4-1. Operadores de JavaScript*

| Operador | Operación                     | A | N | Tipos    |
|----------|-------------------------------|---|---|----------|
| ++       | Previo o posterior al aumento | R | 1 | Ival→num |

|             |  |   |   |                 |
|-------------|--|---|---|-----------------|
| .           |  |   |   |                 |
| --          | Antes o después de la reducción                    | R | 1 | lval→num        |
| -           | Negar el número                                    | R | 1 | num→num         |
| +           | Convertir a número                                 | R | 1 | cualquier→num   |
| ~           | Invertir bits                                      | R | 1 | int→int         |
| !           | Invertir el valor booleano                         | R | 1 | bool→bool       |
| borrar      | Eliminar una propiedad                             | R | 1 | lval→bool       |
| tipo de     | Determinar el tipo de operando                     | R | 1 | cualquier→str   |
| void        | Devuelve un valor indefinido                       | R | 1 | cualquier→undef |
| **          | Exponenciar  | R | 2 | num,num→num     |
| *, /, %     | Multiplicar, dividir, resto                        | L | 2 | num,num→num     |
| +, -        | Sumar, restar                                      | L | 2 | num,num→num     |
| +           | Concatenar cadenas                                 | L | 2 | str,str→str     |
| <<          | Desplazamiento a la izquierda                      | L | 2 | int,int→int     |
| >>          | Desplazamiento a la derecha con extensión de signo | L | 2 | int,int→int     |
| >>>         | Desplazamiento a la derecha con extensión cero     | L | 2 | int,int→int     |
| <, <=,>, >= | Comparar en orden numérico                         | L | 2 | num,num→bool    |
| <, <=,>, >= | Comparar por orden alfabético                      | L | 2 | str,str→bool    |
| de          | Clase de objeto de prueba                          | L | 2 | obj,func→bool   |
| en          | Comprobar si la propiedad existe                   | L | 2 | any,obj→bool    |
| ==          | Prueba de igualdad no estricta                     | L | 2 | any,any→bool    |
| !=          | Prueba de desigualdad no estricta                  | L | 2 | any,any→bool    |
| ==          | Prueba de igualdad estricta                        | L | 2 | any,any→bool    |
| !=          | Prueba de desigualdad estricta                     | L | 2 | any,any→bool    |

|                             |   |    |   |                                   |
|-----------------------------|---|----|---|-----------------------------------|
| &                           | Calcular el AND a nivel de bits               | L  | 2 | int,int→int                       |
| ^                           | Calcular el XOR a nivel de bits               | L  | 2 | int,int→int                       |
|                             | Calcular el OR a nivel de bits                | L  | 2 | int,int→int                       |
| &&                          | Calcular el AND lógico                        | L  | 2 | cualquier,<br>cualquier→cualquier |
|                             | Calcular el OR lógico                         | L  | 2 | cualquier,<br>cualquier→cualquier |
| ??                          | Elija el primer operando<br>definido          | L  | 2 | cualquier,<br>cualquier→cualquier |
| :?                          | Elija el segundo o tercer<br>operando         | R  | 3 | bool,any,any→an y                 |
| =                           | Asignar a una variable o<br>propiedad         | R  | 2 | lval,any→any                      |
| **=, *=, /=, %=,            | Operar y asignarR2lval                        |    |   | ,any→any                          |
| + =, - =, & =, ^ =,<br>  =, |   |    |   |                                   |
| <<=, >>=, >>>=              |   |    |   |                                   |
| ,                           | Descarta el primer operando, devuelve<br>2do. | L2 |   | cualquier,<br>cualquier→cualquier |

#### 4.7.1 Número de operandos

Los operadores pueden clasificarse en función del número de operandos que esperan (su *aridad*). La mayoría de los operadores de JavaScript, como el operador de multiplicación \*, son *operadores binarios* que combinan dos expresiones en una única expresión más compleja. Es decir, esperan dos operandos. JavaScript también admite una serie de operadores *unarios*, que convierten una única expresión en otra más compleja. El operador - en la expresión -x es un operador unario que realiza la operación de negación en el operando x. Por último, JavaScript admite un *operador ternario*, el operador condicional ?:, que combina tres expresiones en una sola.

## 4.7.2 Tipo de operando y resultado

Algunos operadores funcionan con valores de cualquier tipo, pero la mayoría esperan que sus operandos sean de un tipo específico, y la mayoría de los operadores devuelven (o evalúan) un valor de un tipo específico. La columna Tipos de la [Tabla 4-1](#) especifica los tipos de operandos (antes de la flecha) y el tipo de resultado (después de la flecha) para los operadores.

Los operadores de JavaScript suelen convertir el tipo (véase §3.9) de sus operandos según sea necesario. El operador de multiplicación \* espera operandos numéricos, pero la expresión "3" \* "5" es legal porque JavaScript puede convertir los operandos en números. El valor de esta expresión es el número 15, no la cadena "15", por supuesto. Recuerde también que todos los valores de JavaScript son "verdaderos" o "falsos", por lo que los operadores que esperan operandos booleanos funcionarán con un operando de cualquier tipo.

Algunos operadores se comportan de forma diferente según el tipo de los operandos utilizados con ellos. En particular, el operador + suma operandos numéricos pero concatena operandos de cadena. Del mismo modo, los operadores de comparación como < realizan la comparación en orden numérico o alfabético dependiendo del tipo de los operandos. Las descripciones de los operadores individuales explican sus dependencias de tipo y especifican qué conversiones de tipo realizan.

Observe que los operadores de asignación y algunos de los otros operadores enumerados en la [Tabla 4-1](#) esperan un operando de tipo *Ivalue*. *Ivalue* es un término histórico que significa "una expresión que

puede aparecer legalmente en el lado izquierdo de una expresión de asignación". En JavaScript, las variables, las propiedades de los objetos y los elementos de las matrices son lvalues.

### 4.7.3 Efectos secundarios del operador

La evaluación de una expresión simple como `2 * 3` nunca afecta al estado de su programa, y cualquier cálculo futuro que su programa realice no se verá afectado por esa evaluación. Sin embargo, algunas expresiones tienen *efectos secundarios* y su evaluación puede afectar al resultado de futuras evaluaciones. Los operadores de asignación son el ejemplo más obvio: si asignas un valor a una variable o propiedad, eso cambia el valor de cualquier expresión que utilice esa variable o propiedad. Los operadores `++` y `-incremento` y `decremento` son similares, ya que realizan una asignación implícita. El operador de borrado también tiene efectos secundarios: Eliminar una propiedad es como (pero no es lo mismo) asignar `undefined` a la propiedad.

Ningún otro operador de JavaScript tiene efectos secundarios, pero las expresiones de invocación de funciones y de creación de objetos tendrán efectos secundarios si alguno de los operadores utilizados en el cuerpo de la función o del constructor tiene efectos secundarios.

### 4.7.4 Precedencia de los operadores

Los operadores enumerados en [la Tabla 4-1](#) están ordenados de mayor a menor precedencia, con líneas horizontales que separan grupos de operadores del mismo nivel de precedencia. La precedencia de los operadores controla el orden en que se realizan las operaciones. Los operadores con mayor precedencia (más

cercanos a la parte superior de la tabla) se ejecutan antes que los de menor precedencia (más cercanos a la parte inferior).

Considera la siguiente expresión:

```
w = x + y*z;
```

El operador de multiplicación \* tiene mayor precedencia que el operador de suma +, por lo que la multiplicación se realiza antes que la suma. Por otra parte, el operador de asignación = tiene la menor precedencia, por lo que la asignación se realiza después de que se hayan completado todas las operaciones del lado derecho.

La precedencia de los operadores puede anularse con el uso explícito de paréntesis. Para forzar que la suma del ejemplo anterior se realice primero, escriba:

```
w = (x + y)*z;
```

Tenga en cuenta que las expresiones de acceso e invocación de propiedades tienen mayor precedencia que cualquiera de los operadores listados en [la Tabla 4-1](#). Considere esta expresión:

```
// mi es un objeto con una propiedad llamada funciones cuyo valor es un
// array de funciones. Invocamos la función número x, pasándole el argumento // y,
y luego pedimos el tipo del valor devuelto. typeof my.functions[x](y)
```

Aunque typeof es uno de los operadores de mayor prioridad, la operación typeof se realiza sobre el resultado del acceso a la propiedad, el índice del array y la invocación de la función, todos los cuales tienen mayor prioridad que los operadores.

En la práctica, si no está seguro de la precedencia de sus operadores, lo más sencillo es utilizar paréntesis para hacer explícito el orden de evaluación. Las reglas que es importante conocer son las siguientes: la multiplicación y la división se realizan antes que la suma y la resta, y la asignación tiene una precedencia muy baja y casi siempre se realiza en último lugar.

Cuando se añaden nuevos operadores a JavaScript, no siempre encajan de forma natural en este esquema de precedencia. El operador ?? ([§4.13.2](#)) se muestra en la tabla como de menor precedencia que || y &&, pero, de hecho, su precedencia relativa a esos operadores no está definida, y ES2020 requiere que se usen explícitamente paréntesis si se mezcla ?? con || o &&. De manera similar, el nuevo operador de exponentiación \*\* no tiene una precedencia bien definida con respecto al operador de negación unario, y debe usar paréntesis cuando combine la negación con la exponentiación.

#### 4.7.5 Asociatividad de los operadores

En [la Tabla 4-1](#), la columna denominada A especifica la *asociatividad* del operador. Un valor de L especifica la asociatividad de izquierda a derecha, y un valor de R especifica la asociatividad de derecha a izquierda. La asociatividad de un operador especifica el orden en que se realizan las operaciones de la misma precedencia. La asociatividad de izquierda a derecha significa que las operaciones se realizan de izquierda a derecha. Por ejemplo, el operador de sustracción tiene asociatividad de izquierda a derecha, así:

$$w = x - y - z;$$

es lo mismo que:

```
w = ((x - y) - z);
```

Por otro lado, las siguientes expresiones:

```
y = a ** b ** c; x = ~-y;  
w = x = y = z; q = a ? b:c?  
d:e? f:g;
```

son equivalentes:

```
y = (a ** (b ** c)); x = ~(-y);  
w = (x = (y = z)); q = a ? b:(c?  
d:(e? f:g));
```

porque los operadores de exponentiación, unario, asignación y condicional ternario tienen asociatividad de derecha a izquierda.

#### 4.7.6 Orden de evaluación

La precedencia de operadores y la asociatividad especifican el orden en que se realizan las operaciones en una expresión compleja, pero no especifican el orden en que se evalúan las subexpresiones.

JavaScript siempre evalúa las expresiones en un orden estrictamente de izquierda a derecha. En la expresión `w = x + y * z`, por ejemplo, la subexpresión `w` se evalúa en primer lugar, seguida de `x`, `y` y `z`. A continuación, los valores de `y` y `z` se multiplican, se suman al valor de `x` y se asignan a la variable o propiedad especificada por la expresión `w`. La adición de paréntesis a las expresiones puede cambiar el orden relativo de la multiplicación, la suma y la asignación, pero no el orden de evaluación de izquierda a derecha.

El orden de evaluación sólo tiene importancia si alguna de las expresiones que se evalúan tiene efectos secundarios que afectan al valor de otra expresión. Si la expresión `x` incrementa una variable que es utilizada por la expresión `z`, entonces el hecho de que `x` se evalúe antes que `z` es importante.

## 4.8 Expresiones aritméticas

Esta sección cubre los operadores que realizan aritmética u otras manipulaciones numéricas en sus operandos. Los operadores de exponenciación, multiplicación, división y sustracción son sencillos y se tratan primero. El operador de adición recibe una subsección propia porque también puede realizar la concatenación de cadenas y tiene algunas reglas de conversión de tipos inusuales. Los operadores unarios y los operadores a nivel de bits también se tratan en subsecciones propias.

La mayoría de estos operadores aritméticos (excepto los que se indican a continuación) pueden utilizarse con operandos `BigInt` (véase §3.2.5) o con números normales, siempre que no se mezclen los dos tipos.

Los operadores aritméticos básicos son `**` (exponenciación), `*` (multiplicación), `/` (división), `%` (módulo: resto después de la división), `+` (suma) y `-` (resta). Como se ha señalado, hablaremos del operador `+` en una sección propia. Los otros cinco operadores básicos simplemente evalúan sus operandos, convierten los valores en números si es necesario, y luego calculan la potencia, el producto, el cociente, el resto o la diferencia. Los operandos no numéricos que no pueden convertirse en números se convierten

en el valor NaN. Si cualquiera de los operandos es (o se convierte en) NaN, el resultado de la operación es (casi siempre) NaN.

El operador `**` tiene mayor precedencia que `*`, `/` y `%` (que a su vez tienen mayor precedencia que `+` y `-`). A diferencia de los otros operadores, `**` funciona de derecha a izquierda, por lo que `2**2**3` es lo mismo que `2**8`, no `4**3`. Existe una ambigüedad natural en expresiones como `-3**2`. Dependiendo de la precedencia relativa del menos unario y la exponenciación, esa expresión podría significar `(-3)**2` o `-(3**2)`. Los distintos lenguajes manejan esto de forma diferente, y en lugar de elegir un bando, JavaScript simplemente convierte en un error sintáctico omitir los paréntesis en este caso, obligando a escribir una expresión inequívoca. `**` es el operador aritmético más reciente de JavaScript: se añadió al lenguaje con ES2016. El

Sin embargo, la función `Math.pow()` ha estado disponible desde las primeras versiones de JavaScript, y realiza exactamente la misma operación que el operador `**`.

El operador `/` divide su primer operando entre el segundo. Si está acostumbrado a los lenguajes de programación que distinguen entre números enteros y en coma flotante, es posible que espere obtener un resultado entero al dividir un número entero entre otro. En JavaScript, sin embargo, todos los números son de coma flotante, por lo que todas las operaciones de división tienen resultados de coma flotante: `5/2` se evalúa como `2,5`, no como `2`. La división por cero da como resultado un infinito positivo o negativo, mientras que `0/0` se evalúa como NaN: ninguno de estos casos da lugar a un error.

El operador `%` calcula el primer operando en función del segundo.

En otras palabras, devuelve el resto después de la división de números enteros del primer operando por el segundo operando. El signo del resultado es el mismo que el del primer operando. Por ejemplo,  $5 \% 2$  se evalúa como  $1$ , y  $-5 \% 2$  se evalúa como  $-1$ .

Aunque el operador módulo se utiliza normalmente con operandos enteros, también funciona con valores de punto flotante. Por ejemplo,  $6,5 \% 2,1$  se evalúa como  $0,2$ .

#### 4.8.1 El operador +

El operador binario  $+$  añade operandos numéricos o concatena operandos de cadena:

```
1 + 2          //=> 3
"holá" + " " + "allí" //=> "holá allí"
"1" + "2"      //=> "12"
```

Cuando los valores de ambos operandos son números, o ambos son cadenas, entonces es obvio lo que hace el operador  $+$ . En cualquier otro caso, sin embargo, es necesaria la conversión de tipos, y la operación a realizar depende de la conversión realizada. Las reglas de conversión de  $+$  dan prioridad a la concatenación de cadenas: si uno de los operandos es una cadena o un objeto que se convierte en cadena, el otro operando se convierte en cadena y se realiza la concatenación. La suma se realiza sólo si ninguno de los dos operandos es una cadena.

Técnicamente, el operador  $+$  se comporta así:

- Si alguno de sus valores operandos es un objeto, lo convierte en una primitiva utilizando el algoritmo de

conversión de objeto a primitiva descrito en §3.9.3. Los objetos fecha se convierten mediante su método `toString()`, y todos los demás objetos se convierten mediante `valueOf()`, si ese método devuelve un valor primitivo. Sin embargo, la mayoría de los objetos no tienen un método `valueOf()` útil, por lo que también se convierten mediante `toString()`.

- Despues de la conversión de objeto a primitivo, si cualquiera de los dos operandos es una cadena, el otro se convierte en una cadena y se realiza la concatenación.
- En caso contrario, ambos operandos se convierten en números (o en `Nan`) y se realiza la suma.

He aquí algunos ejemplos:

```
1 + 2 // => 3: adición
"1" + "2" // => "12": concatenación
"1" + 2 // => "12": concatenación después de número-a
cadena 1 + {} // => "1[objeto Objeto)": concatenación después de objeto a cadena
true + true // => 2: adición después de booleano a número 2 + null // => 2: la adición
después de null se convierte en 0
2 + undefined // => NaN: la suma después de undefined se convierte en
NaN
```

Por último, es importante señalar que cuando el operador `+` se utiliza con cadenas y números, puede no ser asociativo. Es decir, el resultado puede depender del orden en que se realicen las operaciones.

Por ejemplo:

```
1 + 2 + " ratones ciegos" // => "3 ratones ciegos"
1 + (2 + " ratones ciegos") // => "12 ratones ciegos"
```

La primera línea no tiene paréntesis, y el operador + tiene asociatividad de izquierda a derecha, por lo que los dos números se suman primero, y su suma se concatena con la cadena. En la segunda línea, los paréntesis alteran este orden de operaciones: el número 2 se concatena con la cadena para producir una nueva cadena. A continuación, el número 1 se concatena con la nueva cadena para producir el resultado final.

#### 4.8.2 Operadores aritméticos unarios

Los operadores unarios modifican el valor de un solo operando para producir un nuevo valor. En JavaScript, todos los operadores unarios tienen alta precedencia y son asociativos a la derecha. Los operadores aritméticos unarios descritos en esta sección (+, -, ++ y --) convierten su operando único en un número, si es necesario. Tenga en cuenta que los caracteres de puntuación + y - se utilizan como operadores unarios y binarios.

Los operadores aritméticos unarios son los siguientes:

##### *Unario más (+)*

El operador unario más convierte su operando en un número (o en NaN) y devuelve ese valor convertido. Cuando se utiliza con un operando que ya es un número, no hace nada. Este operador no se puede utilizar con valores BigInt, ya que no se pueden convertir en números normales.

##### *Menos unario (-)*

Cuando se utiliza - como operador unario, convierte su operando en un número, si es necesario, y luego cambia el signo del resultado.

## *Incremento (++)*

El operador `++` incrementa (es decir, suma 1) su único operando, que debe ser un lvalue (una variable, un elemento de una matriz o una propiedad de un objeto). El operador convierte su operando en un número, añade 1 a ese número y asigna el valor incrementado a la variable, elemento o propiedad.

El valor de retorno del operador `++` depende de su posición respecto al operando. Cuando se utiliza antes del operando, donde se conoce como operador de preincremento, incrementa el operando y se evalúa al valor incrementado de ese operando. Cuando se utiliza después del operando, donde se conoce como operador post-incremento, incrementa su operando pero se evalúa al valor *no incrementado de ese operando*. Considere la diferencia entre estas dos líneas de código:

```
let i = 1, j = ++i; // i y j son ambos 2 let n = 1, m = n++; // n es 2, m  
es 1
```

Tenga en cuenta que la expresión `x++` no es siempre la misma que `x=x+1`. El operador `++` nunca realiza una concatenación de cadenas: siempre convierte su operando en un número y lo incrementa. Si `x` es la cadena "1", `++x` es el número 2, pero `x+1` es la cadena "11".

Tenga en cuenta también que, debido a la inserción automática de punto y coma de JavaScript, no puede insertar un salto de línea entre el operador de post-incremento y el operando que lo precede. Si lo hace, JavaScript tratará el operando como una sentencia completa por sí misma e insertará un punto y coma antes de ella.

Este operador, tanto en su forma de pre como de postincremento, se utiliza habitualmente para incrementar un contador que controla un bucle `for` ([§5.4.3](#)).

### *Disminución (--)*

El operador -- espera un operando lvalue. Convierte el valor del operando en un número, resta 1 y asigna el valor decrementado de nuevo al operando. Al igual que el operador ++, el valor de retorno de -- depende de su posición respecto al operando. Cuando se utiliza antes del operando, disminuye y devuelve el valor disminuido. Cuando se utiliza después del operando, decrementa el operando pero devuelve el valor *no decrementado*. Cuando se utiliza después del operando, no se permite ningún salto de línea entre el operando y el operador.

### **4.8.3 Operadores Bitwise**

Los operadores "bitwise" realizan una manipulación de bajo nivel de los bits en la representación binaria de los números. Aunque no realizan operaciones aritméticas tradicionales, se clasifican aquí como operadores aritméticos porque operan con operandos numéricos y devuelven un valor numérico. Cuatro de estos operadores realizan álgebra booleana sobre los bits individuales de los operandos, comportándose como si cada bit de cada operando fuera un valor booleano (1=verdadero, 0=falso). Los otros tres operadores bitwise se utilizan para desplazar los bits a la izquierda y a la derecha. Estos operadores no se utilizan habitualmente en la programación de JavaScript, y si no estás familiarizado con la representación binaria de los enteros, incluida la representación del complemento a dos de los enteros negativos, probablemente puedas saltarte esta sección.

Los operadores bitácora esperan operandos enteros y se comportan como si esos valores estuvieran representados como enteros de 32 bits en lugar de valores de punto flotante de 64 bits. Estos operadores convierten sus operandos en números, si es

necesario, y luego coaccionan los valores numéricos a enteros de 32 bits eliminando cualquier parte fraccionaria y cualquier bit más allá del 32. Los operadores de desplazamiento requieren un operando del lado derecho entre 0 y 31. Después de convertir este operando en un entero de 32 bits sin signo, eliminan cualquier bit más allá del 5º, lo que produce un número en el rango apropiado. Sorprendentemente, NaN, Infinity y -Infinity se convierten en 0 cuando se utilizan como operandos de estos operadores a nivel de bits.

Todos estos operadores a nivel de bits, excepto `>>>`, pueden utilizarse con operandos numéricos normales o con operandos BigInt (véase §3.2.5).

### *AND (&) a nivel de bits*

El operador `&` realiza una operación booleana AND en cada bit de sus argumentos enteros. Un bit se establece en el resultado sólo si el bit correspondiente está establecido en ambos operandos. Por ejemplo, `0x1234 & 0x00FF` equivale a `0x0034`.

### *O a nivel de bits (|)*

El operador `|` realiza una operación booleana OR en cada bit de sus argumentos enteros. Un bit se establece en el resultado si el bit correspondiente está establecido en uno o ambos operandos. Por ejemplo, `0x1234 | 0x00FF` se evalúa como `0x12FF`.

### *XOR a nivel de bit (^)*

El operador `^` realiza una operación booleana OR exclusiva en cada bit de sus argumentos enteros. El operador OR exclusivo significa que el operando uno es verdadero o el operando dos es verdadero, pero no ambos. Un bit se activa en el resultado de

esta operación si el bit correspondiente está activado en uno (pero no en ambos) de los dos operandos. Por ejemplo,  $0xFF00 \wedge 0xF0F0$  se evalúa como  $0x0FF0$ .

### *Bitwise NOT ( $\sim$ )*

El operador  $\sim$  es un operador unario que aparece antes de su único operando entero. Funciona invirtiendo todos los bits del operando. Debido a la forma en que se representan los enteros con signo en JavaScript, aplicar el operador  $\sim$  a un valor equivale a cambiar su signo y restar 1. Por ejemplo,  $\sim 0x0F$  se evalúa como  $0xFFFFF0$ , es decir, -16.

### *Desplazamiento a la izquierda (<<)*

El operador  $<<$  desplaza todos los bits de su primer operando hacia la izquierda el número de posiciones especificado en el segundo operando, que debe ser un entero entre 0 y 31. Por ejemplo, en la operación  $a << 1$ , el primer bit (el bit de los unos) de  $a$  se convierte en el segundo bit (el bit de los dos), el segundo bit de  $a$  se convierte en el tercero, etc. Se utiliza un cero para el nuevo primer bit, y se pierde el valor del bit 32. Desplazar un valor hacia la izquierda una posición equivale a multiplicar por 2, desplazar dos posiciones equivale a multiplicar por 4, y así sucesivamente. Por ejemplo,  $7 << 2$  se evalúa como 28.

### *Desplazamiento a la derecha con signo (>>)*

El operador  $>>$  desplaza todos los bits de su primer operando a la derecha el número de posiciones especificado en el segundo operando (un entero entre 0 y 31). Los bits desplazados a la derecha se pierden. Los bits rellenados a la izquierda dependen del bit de signo del operando original, para preservar el signo del resultado. Si el primer operando es positivo, el resultado tiene ceros en los bits altos; si el primer operando es negativo, el

resultado tiene unos en los bits altos. Desplazar un valor positivo a la derecha un lugar es equivalente a dividir por 2 (descartando el resto), desplazar a la derecha dos lugares es equivalente a una división entera por 4, y así sucesivamente.  $7 >> 1$  se evalúa como 3, por ejemplo, pero tenga en cuenta que  $-7 >> 1$  se evalúa como -4.

#### *Desplazamiento a la derecha con relleno cero (>>>)*

El operador `>>>` es igual que el operador `>>`, salvo que los bits desplazados a la izquierda son siempre cero, independientemente del signo del primer operando. Esto es útil cuando se quiere tratar valores de 32 bits con signo como si fueran enteros sin signo.  $-1 >> 4$  se evalúa como -1, pero  $-1 >>> 4$  se evalúa como `0xFFFF`, por ejemplo. Este es el único de los operadores a nivel de bits de JavaScript que no puede utilizarse con valores `BigInt`. `BigInt` no representa los números negativos poniendo el bit alto de la forma en que lo hacen los enteros de 32 bits, y este operador sólo tiene sentido para esa representación particular de complemento a dos.

## 4.9 Expresiones relacionales

Esta sección describe los operadores relacionales de JavaScript. Estos operadores comprueban la existencia de una relación (como "igual", "menor que" o "propiedad de") entre dos valores y devuelven verdadero o falso dependiendo de si existe esa relación. Las expresiones relacionales siempre se evalúan a un valor booleano, y ese valor se utiliza a menudo para controlar el flujo de ejecución del programa en las sentencias `if`, `while` y `for` (véase [el capítulo 5](#)). Las subsecciones siguientes documentan los operadores de igualdad y desigualdad, los operadores de comparación y los otros dos operadores relacionales de JavaScript, `in` e `instanceof`.

## 4.9.1 Operadores de igualdad y desigualdad

Los operadores `==` y `===` comprueban si dos valores son iguales, utilizando dos definiciones diferentes de igualdad. Ambos operadores aceptan operandos de cualquier tipo, y ambos devuelven verdadero si sus operandos son iguales y falso si son diferentes. El operador `===` se conoce como operador de igualdad estricta (o a veces operador de identidad), y comprueba si sus dos operandos son "idénticos" utilizando una definición estricta de igualdad. El operador `==` se conoce como operador de igualdad; comprueba si sus dos operandos son "iguales" utilizando una definición más relajada de igualdad que permite conversiones de tipo.

Los operadores `!=` y `!==` comprueban exactamente lo contrario de los operadores `==` y `===`. El operador de desigualdad `!=` devuelve falso si dos valores son iguales entre sí según `==` y devuelve verdadero en caso contrario. El operador `!==` devuelve falso si dos valores son estrictamente iguales entre sí y devuelve verdadero en caso contrario. Como se verá en §4.10, el operador `!` calcula la operación booleana NOT. Esto hace que sea fácil recordar que `!=` y `!==` significan "no igual a" y "no estrictamente igual a".

### EL `=`, `==` Y `===` OPERADORES

JavaScript soporta `=`, `==` y `===` operadores. Asegúrese de entender las diferencias entre estos operadores de asignación, igualdad e igualdad estricta, y tenga cuidado de utilizar el correcto al codificar. Aunque es tentador leer los tres operadores como "iguales", puede ayudar a reducir la confusión si

léase "obtiene" o "se le asigna" → `==` para "es estrictamente igual a" → `==`.

El `==` es una característica heredada de JavaScript y es ampliamente considerada como una fuente de errores. Casi siempre hay que utilizar `===` en lugar de `==` y `!==` en lugar de `!=`.

Como se mencionó en §3.8, los objetos de JavaScript se comparan por referencia, no por valor. Un objeto es igual a sí mismo, pero no a cualquier otro objeto. Si dos objetos distintos tienen el mismo número de propiedades, con los mismos nombres y valores, siguen sin ser iguales. Del mismo modo, dos matrices que tienen los mismos elementos en el mismo orden no son iguales entre sí.

## IGUALDAD ESTRICTA

El operador de igualdad estricta === evalúa sus operandos y luego compara los dos valores de la siguiente manera, sin realizar ninguna conversión de tipo:

- Si los dos valores tienen tipos diferentes, no son iguales.
- Si ambos valores son nulos o ambos valores son indefinidos, son iguales.
- Si ambos valores son el valor booleano verdadero o ambos son el valor booleano falso, son iguales.
- Si uno o ambos valores son NaN, no son iguales. (Esto es sorprendente, pero el valor NaN nunca es igual a ningún otro valor, ¡incluso él mismo! Para comprobar si un valor  $x$  es NaN, utilice  $x !== x$ , o la función global isNaN()).
- Si ambos valores son números y tienen el mismo valor, son iguales. Si un valor es 0 y el otro es -0, también son iguales.
- Si ambos valores son cadenas y contienen exactamente los mismos valores de 16 bits (véase la barra lateral en §3.3) en las mismas posiciones, son iguales. Si las cadenas difieren en longitud o contenido, no son iguales. Dos cadenas pueden tener el mismo significado y la misma apariencia visual, pero

aún así estar codificadas usando diferentes secuencias de valores de 16 bits. JavaScript no realiza ninguna normalización Unicode, y un par de cadenas como ésta no se considera igual a los operadores === o ==.

- Si ambos valores se refieren al mismo objeto, matriz o función, son iguales. Si se refieren a objetos diferentes, no son iguales, aunque ambos objetos tengan propiedades idénticas.

## IGUALDAD CON CONVERSIÓN DE TIPOS

El operador de igualdad == es como el operador de igualdad estricta, pero es menos estricto. Si los valores de los dos operandos no son del mismo tipo, intenta algunas conversiones de tipo y vuelve a intentar la comparación:

- Si los dos valores tienen el mismo tipo, compruebe su igualdad estricta como se ha descrito anteriormente. Si son estrictamente iguales, son iguales. Si no son estrictamente iguales, no lo son.
- Si los dos valores no tienen el mismo tipo, el operador == puede seguir considerándolos iguales. Utiliza las siguientes reglas y conversiones de tipo para comprobar la igualdad:
  - Si un valor es nulo y el otro es indefinido, son iguales.
  - Si un valor es un número y el otro es una cadena, convierta la cadena en un número e intente la comparación de nuevo, utilizando el valor convertido.
  - Si cualquiera de los dos valores es verdadero, conviértalo en 1 e intente la comparación de nuevo.

Si cualquiera de los dos valores es falso, conviértalo en 0 e intente la comparación de nuevo.

- Si un valor es un objeto y el otro es un número o una cadena, convierta el objeto en una primitiva utilizando el algoritmo descrito en §3.9.3 y vuelva a intentar la comparación. Un objeto se convierte en un valor primitivo mediante su método `toString()` o su método `valueOf()`. Las clases incorporadas del núcleo de JavaScript intentan la conversión `valueOf()` antes de la conversión `toString()`, excepto la clase Date, que realiza la conversión `toString()`.
- Cualquier otra combinación de valores no es igual.

Como ejemplo de comprobación de la igualdad, considere la comparación:

```
"1" == true //=> true
```

Esta expresión se evalúa como verdadera, lo que indica que estos valores de aspecto tan diferente son de hecho iguales. El valor booleano "true" se convierte primero en el número 1, y la comparación se realiza de nuevo. A continuación, la cadena "1" se convierte en el número 1. Como ambos valores son ahora iguales, la comparación devuelve true.

#### 4.9.2 Operadores de comparación

Los operadores de comparación comprueban el orden relativo (numérico o alfabético) de sus dos operandos:

*Menos de (<)*

El operador `<` se evalúa como verdadero si su primer operando es menor que su segundo operando; en caso contrario, se evalúa como falso.

#### *Mayor que (`>`)*

El operador `>` se evalúa como verdadero si su primer operando es mayor que su segundo operando; en caso contrario, se evalúa como falso.

#### *Menor o igual (`<=`)*

El operador `<=` se evalúa como verdadero si su primer operando es menor o igual que su segundo operando; en caso contrario, se evalúa como falso.

#### *Mayor o igual (`>=`)*

El operador `>=` se evalúa como verdadero si su primer operando es mayor o igual que su segundo operando; en caso contrario, se evalúa como falso.

Los operandos de estos operadores de comparación pueden ser de cualquier tipo. Sin embargo, la comparación sólo puede realizarse con números y cadenas, por lo que los operandos que no son números o cadenas se convierten.

La comparación y la conversión se producen de la siguiente manera:

- Si cualquiera de los operandos se evalúa como un objeto, ese objeto se convierte en un valor primitivo, como se describe al final de §3.9.3; si su método `valueOf()` devuelve un valor primitivo, se utiliza ese valor. En caso

contrario, se utiliza el valor devuelto por su método `toString()`.

- Si, después de cualquier conversión requerida de objeto a primitivo, ambos operandos son cadenas, las dos cadenas se comparan, utilizando el orden alfabético, donde el "orden alfabético" se define por el orden numérico de los valores Unicode de 16 bits que componen las cadenas.
- Si, tras la conversión de objeto a primitivo, al menos un operando no es una cadena, ambos operandos se convierten en números y se comparan numéricamente. 0 y -0 se consideran iguales. Infinito es mayor que cualquier número que no sea él mismo, e Infinito es menor que cualquier número que no sea él mismo. Si cualquiera de los operandos es (o se convierte en) NaN, el operador de comparación siempre devuelve false. Aunque los operadores aritméticos no permiten mezclar valores `BigInt` con números normales, los operadores de comparación sí permiten comparar números con `BigInts`.

Recuerde que las cadenas de JavaScript son secuencias de valores enteros de 16 bits, y que la comparación de cadenas es sólo una comparación numérica de los valores de las dos cadenas. El orden de codificación numérica definido por Unicode puede no coincidir con el orden de cotejo tradicional utilizado en un idioma o una localidad en particular. Tenga en cuenta, en particular, que la comparación de cadenas distingue entre mayúsculas y minúsculas, y que todas las letras ASCII mayúsculas son "menos" que todas las letras ASCII minúsculas. Esta regla puede causar resultados confusos si no se espera. Por ejemplo, según el operador `<`, la cadena "Zoo" está antes que la cadena "aardvark".

Para un algoritmo de comparación de cadenas más robusto, pruebe el método `String.localeCompare()`, que también tiene en cuenta las definiciones de orden alfabético específicas de cada localidad. Para las comparaciones entre mayúsculas y minúsculas, puede convertir las cadenas en minúsculas o en mayúsculas utilizando `String.toLowerCase()` o `String.toUpperCase()`. Y, para que sea más general y mejor herramienta de comparación de cadenas localizada, utilice la clase `Intl.Collator` descrita en [§11.7.3](#).

Tanto el operador `+` como los operadores de comparación se comportan de forma diferente para los operandos numéricos y los de cadena. El operador `+` favorece a las cadenas: realiza la concatenación si cualquiera de los dos operandos es una cadena. Los operadores de comparación favorecen a los números y sólo realizan la comparación de cadenas si ambos operandos son cadenas:

```
1 + 2 // => 3: adición.  
"1" + "2" // => "12": concatenación.  
"1" + 2 // => "12": 2 se convierte en "2".  
11 < 3 // => falso: comparación numérica.  
"11" < "3" // => true: comparación de cadenas.  
"11" < 3 // => falso: comparación numérica, "11" convertido a 11. "uno" < 3 // => falso: comparación numérica, "uno" convertido a NaN.
```

Por último, tenga en cuenta que los operadores `<=` (menor que o igual) y `>=` (mayor que o igual) no se basan en los operadores de igualdad o igualdad estricta para determinar si dos valores son "iguales". En su lugar, el operador menor que o igual se define simplemente como "no mayor que", y el operador mayor que o igual se define como "no menor que". La única excepción ocurre cuando

uno de los operandos es (o se convierte en) NaN, en cuyo caso, los cuatro operadores de comparación devuelven falso.

### 4.9.3 El operador in

El operador in espera un operando del lado izquierdo que sea una cadena, un símbolo o un valor que pueda convertirse en una cadena. Espera un operando del lado derecho que es un objeto. Se evalúa como verdadero si el valor del lado izquierdo es el nombre de una propiedad del objeto del lado derecho. Por ejemplo:

```
let punto = {x: 1, y: 1}; // Definir un objeto
"x" en el punto // => verdadero: el objeto tiene la propiedad "x" "z" en el punto // =>
falso: el objeto no tiene la propiedad "z".
"toString" en el punto // => true: el objeto hereda el método toString

let data = [7,8,9]; // Un array con elementos
(indices) 0, 1 y 2 "0" en los datos // => true: el array tiene un elemento
"0"
1 en datos // => verdadero: los números se convierten en cadenas 3 en datos // =>
falso: no hay elemento 3
```

### 4.9.4 El operador instanceof

El operador instanceof espera un operando del lado izquierdo que es un objeto y un operando del lado derecho que identifica una clase de objetos. El operador evalúa a true si el objeto del lado izquierdo es una instancia de la clase del lado derecho y evalúa a false en caso contrario. [El capítulo 9](#) explica que, en JavaScript, las clases de objetos se definen mediante la función constructora que los inicializa. Por lo tanto, el operando del lado derecho de instanceof debe ser una función. He aquí algunos ejemplos:

```
let d = new Date(); // Crear un nuevo objeto con el constructor Date() d instanceof Date  
// => true: d fue creado con Date() d instanceof Object // => true: todos los objetos son  
instancias de  
Objeto  
d instanceof Number // => false: d no es un objeto Number let a = [1, 2, 3]; // Crea  
un array con la sintaxis literal de array a instanceof Array // => true: a es un array a  
instanceof Object // => true: todos los arrays son objetos a instanceof RegExp // =>  
false: los arrays no son expresiones regulares
```

Tenga en cuenta que todos los objetos son instancias de Object. instanceof considera las "superclases" cuando decide si un objeto es una instancia de una clase. Si el operando del lado izquierdo de instanceof no es un objeto, instanceof devuelve false. Si el operando del lado derecho no es una clase de objetos, lanza un TypeError.

Para entender cómo funciona el operador instanceof, debes entender la "cadena de prototipos". Este es el mecanismo de herencia de JavaScript, y se describe en §6.3.2. Para evaluar la expresión o instanceof f, JavaScript evalúa f.prototype, y luego busca ese valor en la cadena de prototipos de o. Si lo encuentra, entonces o es una instancia de f (o de una subclase de f) y el operador devuelve true. Si f.prototype no es uno de los valores en la cadena de prototipos de o, entonces o no es una instancia de f y instanceof devuelve false.

## 4.10 Expresiones lógicas

Los operadores lógicos `&&`, `||` y `!` realizan álgebra booleana y se utilizan a menudo junto con los operadores relacionales para combinar dos expresiones relacionales en una expresión más

compleja. Estos operadores se describen en las subsecciones siguientes. Para entenderlos completamente, es posible que desee revisar el concepto de valores "verdaderos" y "falsos" introducido en [§3.4](#).

#### 4.10.1 AND lógico (`&&`)

El operador `&&` puede entenderse en tres niveles diferentes. En el nivel más sencillo, cuando se utiliza con operandos booleanos, `&&` realiza la operación booleana AND sobre los dos valores: devuelve verdadero si y sólo si su primer operando y su segundo operando son verdaderos. Si uno o ambos operandos son falsos, devuelve falso.

`&&` se utiliza a menudo como conjunción para unir dos expresiones relacionales:

```
x === 0 && y === 0 // verdadero si, y sólo si, x e y son ambos 0
```

Las expresiones relacionales siempre se evalúan como verdadero o falso, por lo que cuando se utilizan de esta manera, el operador `&&` devuelve por sí mismo verdadero o falso. Los operadores relacionales tienen mayor precedencia que `&&` (`y ||`), por lo que expresiones como ésta pueden escribirse sin paréntesis.

Pero `&&` no requiere que sus operandos sean valores booleanos. Recuerde que todos los valores de JavaScript son "verdaderos" o "falsos". (Véase [§3.4](#) para más detalles).

Los valores falsos son `false`, `null`, `undefined`, `0`, `-0`, `NaN` y `""`. Todos los demás valores, incluidos todos los objetos, son verdaderos). El segundo nivel en el que se puede entender `&&` es como un operador booleano AND para valores verdaderos y falsos. Si ambos operandos son verdaderos, el operador devuelve un valor

verdadero. En caso contrario, uno o ambos operandos deben ser falsos, y el operador devuelve un valor falso. En JavaScript, cualquier expresión o sentencia que espere un valor booleano funcionará con un valor verdadero o falso, por lo que el hecho de que `&&` no devuelva siempre verdadero o falso no causa problemas prácticos.

Observe que esta descripción dice que el operador devuelve "un valor verdadero" o "un valor falso", pero no especifica cuál es ese valor. Para ello, necesitamos describir `&&` en el tercer y último nivel. Este operador comienza evaluando su primer operando, la expresión de la izquierda. Si el valor de la izquierda es falso, el valor de toda la expresión también debe ser falso, así que `&&` simplemente devuelve el valor de la izquierda y ni siquiera evalúa la expresión de la derecha.

Por otra parte, si el valor de la izquierda es verdadero, el valor global de la expresión depende del valor de la derecha. Si el valor de la derecha es verdadero, el valor total debe ser verdadero, y si el valor de la derecha es falso, el valor total debe ser falso. Así que cuando el valor de la izquierda es verdadero, el operador `&&` evalúa y devuelve el valor de la derecha:

`let o = {x: 1}; let p = null; o && o.x // => 1: o es verdadero, así que devuelve el valor de o.x p && p.x // => null: p es falso, así que lo devuelve y no evalúa p.x`

Es importante entender que `&&` puede o no evaluar su operando derecho. En este ejemplo de código, la variable `p` es nula, y la expresión `p.x`, si se evaluara, causaría un `TypeError`. Pero el código utiliza `&&` de forma idiomática para que `p.x` se evalúe sólo si `p` es verdadera-no nula o indefinida.

El comportamiento de `&&` se denomina a veces cortocircuito, y a veces puede ver código que explota a propósito este comportamiento para ejecutar código condicionalmente. Por ejemplo, las siguientes dos líneas de código JavaScript tienen efectos equivalentes:

```
if (a === b) stop(); // Invoca stop() sólo si a === b (a === b) && stop(); // Esto hace lo mismo
```

En general, debe tener cuidado siempre que escribe una expresión con efectos secundarios (asignaciones, incrementos, decrementos o invocaciones de funciones) en el lado derecho de `&&`. El que se produzcan esos efectos secundarios depende del valor del lado izquierdo.

A pesar de la forma algo compleja en que este operador funciona en realidad, se utiliza más comúnmente como un simple operador de álgebra booleana que funciona en valores verdaderos y falsos.

#### 4.10.2 OR lógico (`||`)

El operador `||` realiza la operación booleana OR sobre sus dos operandos. Si uno o ambos operandos son verdaderos, devuelve un valor verdadero. Si ambos operandos son falsos, devuelve un valor falso.

Aunque el operador `||` se utiliza más a menudo como operador booleano OR, al igual que el operador `&&`, tiene un comportamiento más complejo. Comienza evaluando su primer operando, la expresión de su izquierda. Si el valor de este primer operando es verdadero, hace un cortocircuito y devuelve ese valor verdadero sin evaluar nunca la expresión de la derecha. Si, por el contrario, el valor

•  
del primer operando es falso, entonces `||` evalúa su segundo operando y devuelve el valor de esa expresión.

Al igual que con el operador `&&`, debe evitar los operandos del lado derecho que incluyan efectos secundarios, a menos que quiera utilizar a propósito el hecho de que la expresión del lado derecho no pueda ser evaluada.

Un uso idiomático de este operador es seleccionar el primer valor verdadero de un conjunto de alternativas:

```
// Si maxWidth es verdadero, úsalo. Si no, busca un valor en // el objeto de
preferencias. Si no es verdadero, usa una constante codificada. let max =
maxWidth || preferences.maxWidth | 500;
```

Tenga en cuenta que si 0 es un valor legal para `maxWidth`, este código no funcionará correctamente, ya que 0 es un valor falso. Véase el operador `??` ([§4.13.2](#)) para una alternativa.

Antes de ES6, este modismo se utilizaba a menudo en las funciones para proporcionar valores por defecto a los parámetros:

```
// Copiar las propiedades de o a p, y devolver p
function copy(o, p) { p = p || {}; // Si
no se pasa ningún objeto para p, se utiliza un objeto recién creado. // El cuerpo de
la función va aquí
}
```

En ES6 y posteriores, sin embargo, este truco ya no es necesario porque el valor del parámetro por defecto podría escribirse simplemente en la propia definición de la función: `function copy(o, p={}) { ... }`.

### 4.10.3 Lógica NOT (!)

El operador `!` es un operador unario; se coloca delante de un solo operando. Su propósito es invertir el valor booleano de su operando. Por ejemplo, si `x` es verdadero, `!x` se evalúa como falso. Si `x` es falso, entonces `!x` es verdadero.

A diferencia de los operadores `&&` y `||`, el operador `!` convierte su operando en un valor booleano (utilizando las reglas descritas en el Capítulo 3) antes de invertir el valor convertido. Esto significa que `!` siempre devuelve verdadero o falso y que se puede convertir cualquier valor `x` en su valor booleano equivalente aplicando este operador dos veces: `!!x` (véase §3.9.2).

Como operador unario, `!` tiene alta precedencia y se vincula fuertemente. Si desea invertir el valor de una expresión como `p && q`, debe utilizar paréntesis: `!(p && q)`. Vale la pena señalar dos leyes del álgebra booleana que podemos expresar utilizando la sintaxis de JavaScript:

```
// Las leyes de DeMorgan! (p && q) === (! p | ! q) // => verdadero: para todos los
valores de p y q ! (p | q) === (! p && ! q) // => verdadero: para todos los valores de p y
q
```

## 4.11 Expresiones de asignación

JavaScript utiliza el operador `=` para asignar un valor a una variable o propiedad. Por ejemplo:

```
i = 0; // Poner la variable i a 0.
o. x = 1; // Poner la propiedad x del objeto o a 1.
```

El operador `=` espera que su operando del lado izquierdo sea un lvalue: una variable o propiedad de un objeto (o elemento de un

array). Espera que su operando derecho sea un valor arbitrario de cualquier tipo. El valor de una expresión de asignación es el valor del operando derecho. Como efecto secundario, el operador = asigna el valor de la derecha a la variable o propiedad de la izquierda para que las futuras referencias a la variable o propiedad se evalúen al valor.

Aunque las expresiones de asignación suelen ser bastante sencillas, a veces puede ver el valor de una expresión de asignación utilizado como parte de una expresión mayor. Por ejemplo, puede asignar y probar un valor en la misma expresión con un código como éste:

```
(a = b) === 0
```

Si lo hace, asegúrese de tener clara la diferencia entre el =

y === operadores! Tenga en cuenta que = tiene una precedencia muy baja, y los paréntesis suelen ser necesarios cuando el valor de una asignación se va a utilizar en una expresión mayor.

El operador de asignación tiene asociatividad de derecha a izquierda, lo que significa que cuando aparecen varios operadores de asignación en una expresión, se evalúan de derecha a izquierda. Por lo tanto, puedes escribir código como éste para asignar un único valor a múltiples variables:

```
i = j = k = 0; // Inicializar 3 variables a 0
```

#### 4.11.1 Asignación con operación

Además del operador de asignación normal =, JavaScript admite otros operadores de asignación que proporcionan atajos al combinar la

asignación con alguna otra operación. Por ejemplo, el operador `+=` realiza una suma y una asignación. La siguiente expresión:

```
total += salesTax;
```

es equivalente a éste:

```
total = total + salesTax;
```

Como es de esperar, el operador `+=` funciona para números o cadenas.

Para los operandos numéricos, realiza la suma y la asignación; para los operandos de cadena, realiza la concatenación y la asignación.

Otros operadores similares son `-=`, `*=`, `&=`, etc. [La Tabla 4-2](#) los enumera todos.

*Tabla 4-2. Operadores de asignación*

| Operador                   | Ejemplo                        | Equivalente                       |
|----------------------------|--------------------------------|-----------------------------------|
| <code>+=</code>            | <code>a += b</code>            | <code>a = a + b</code>            |
| <code>-=</code>            | <code>a -= b</code>            | <code>a = a - b</code>            |
| <code>*=</code>            | <code>a *= b</code>            | <code>a = a * b</code>            |
| <code>/=</code>            | <code>a /= b</code>            | <code>a = a / b</code>            |
| <code>%=</code>            | <code>a %= b</code>            | <code>a = a % b</code>            |
| <code>**=</code>           | <code>a **= b</code>           | <code>a = a ** b</code>           |
| <code>&lt;&lt;=</code>     | <code>a &lt;&lt;= b</code>     | <code>a = a &lt;&lt; b</code>     |
| <code>&gt;&gt;=</code>     | <code>a &gt;&gt;= b</code>     | <code>a = a &gt;&gt; b</code>     |
| <code>&gt;&gt;&gt;=</code> | <code>a &gt;&gt;&gt;= b</code> | <code>a = a &gt;&gt;&gt; b</code> |

|                     |                         |                            |
|---------------------|-------------------------|----------------------------|
| <code>&amp;=</code> | <code>a &amp;= b</code> | <code>a = a &amp; b</code> |
| <code> =</code>     | <code>a  = b</code>     | <code>a = a   b</code>     |
| <code>^=</code>     | <code>a ^= b</code>     | <code>a = a ^ b</code>     |

En la mayoría de los casos, la expresión:

`a op= b`

donde op es un operador, es equivalente a la expresión

`a = a op b`

En la primera línea, la expresión a se evalúa una vez. En la segunda, se evalúa dos veces. Los dos casos difieren sólo si a incluye efectos secundarios como una llamada a una función o un operador de incremento. Las dos asignaciones siguientes, por ejemplo, no son iguales:

```
datos[i++] *= 2; datos[i++] =
datos[i++] * 2;
```

## 4.12 Expresiones de evaluación

Como muchos lenguajes interpretados, JavaScript tiene la capacidad de interpretar cadenas de código fuente de JavaScript, evaluándolas para producir un valor.

JavaScript hace esto con la función global eval():

```
eval("3+2") //=> 5
```

La evaluación dinámica de cadenas de código fuente es una potente característica del lenguaje que casi nunca es necesaria en la práctica. Si se encuentra utilizando eval(), debería pensar cuidadosamente si

realmente necesita utilizarla. En particular, eval() puede ser un agujero de seguridad, y nunca debe pasar ninguna cadena derivada

eval() es una función, pero se incluye en este capítulo sobre expresiones porque realmente debería haber sido un operador. Las primeras versiones del lenguaje definieron una función eval(), y desde entonces, los diseñadores del lenguaje y los escritores de intérpretes han ido poniendo restricciones en ella que la hacen cada vez más parecida a un operador. Los intérpretes modernos de JavaScript realizan mucho análisis y optimización del código. En general, si una función llama a eval(), el intérprete no puede optimizar esa función. El problema de definir eval() como una función es que puede recibir otros nombres:

```
dejar f = eval; dejar g =
f;
```

Si esto se permite, el intérprete no puede saber con seguridad qué funciones llaman a eval(), por lo que no puede optimizar de forma agresiva. Este problema podría haberse evitado si eval() fuera un operador (y una palabra reservada). Aprenderemos (en §4.12.2 y §4.12.3) sobre las restricciones impuestas a eval() para hacerla

más como operador.

de la entrada del usuario a eval(). Con un lenguaje tan complicado como JavaScript, no hay manera de sanear la entrada del usuario para que sea seguro utilizarla con eval(). Debido a estos problemas de seguridad, algunos servidores web utilizan la cabecera HTTP "Content-SecurityPolicy" para desactivar eval() en todo el sitio web.

Las subsecciones que siguen explican el uso básico de eval() y explican dos versiones restringidas del mismo que tienen menos impacto en el optimizador.

### ¿ES EVAL() UNA FUNCIÓN O UN OPERADOR?

### **4.12.1 eval()**

eval() espera un argumento. Si se pasa cualquier valor que no sea una cadena, simplemente devuelve ese valor. Si se pasa una cadena, intenta analizar la cadena como código JavaScript, lanzando un SyntaxError si falla. Si analiza la cadena con éxito, evalúa el código y devuelve el valor de la última expresión o sentencia de la cadena o undefined si la última expresión o sentencia no tiene valor. Si la cadena evaluada lanza una excepción, esa excepción se propaga desde la llamada a eval().

La clave de eval() (cuando se invoca así) es que utiliza el entorno de variables del código que lo llama. Es decir, busca los valores de las variables y define nuevas variables y funciones de la misma manera que lo hace el código local. Si una función define una variable local x y luego llama a eval("x"), obtendrá el valor de la variable local. Si llama a eval("x=1"), cambia el valor de la variable local. Y si la función llama a eval("var y = 3;"), declara una nueva variable local y. En cambio, si la cadena evaluada utiliza let o const, la variable o constante declarada será local a la evaluación y no estará definida en el entorno de llamada.

Del mismo modo, una función puede declarar una función local con un código como este:

```
eval("function f() { return x+1; }");
```

Si llamas a eval() desde el código de nivel superior, opera sobre variables y funciones globales, por supuesto.

Tenga en cuenta que la cadena de código que pase a eval() debe tener sentido sintáctico por sí misma: no puede utilizarla para pegar fragmentos de código en una función. No tiene sentido escribir eval("return;"), por ejemplo, porque return sólo es legal dentro de las funciones, y el hecho de que la cadena evaluada utilice el mismo entorno de variables que la función que la llama no la hace parte de esa función. Si su cadena tiene sentido como un script independiente (incluso uno muy corto como x=0 ), es legal pasarlo a eval(). De lo contrario, eval() lanzará un SyntaxError.

## 4.12.2 Global eval()

Es la capacidad de eval() de cambiar las variables locales lo que resulta tan problemático para los optimizadores de JavaScript. Sin embargo, como solución, los intérpretes simplemente hacen menos optimización en cualquier función que llame a eval(). Sin embargo, ¿qué debe hacer un intérprete de JavaScript si un script define un alias para eval() y luego llama a esa función con otro nombre? La especificación de JavaScript declara que cuando eval() es invocada por cualquier nombre que no sea "eval", debe evaluar la cadena como si fuera código global de alto nivel. El código evaluado puede definir nuevas variables o funciones globales, y puede establecer variables globales, pero no utilizará ni modificará

ninguna variable local de la función que llama, y por tanto no interferirá con las optimizaciones locales.

Una "evaluación directa" es una llamada a la función eval() con una expresión que utiliza el nombre exacto y no cualificado "eval" (que empieza a parecer una palabra reservada). Las llamadas directas a eval() utilizan el entorno de variables del contexto de llamada.

Cualquier otra llamada -una llamada indirecta- utiliza el objeto global como su entorno de variables y no puede leer, escribir o definir variables o funciones locales. (Tanto las llamadas directas como las indirectas pueden definir nuevas variables sólo con var. Los usos de let y const dentro de una cadena evaluada crean variables y constantes que son locales a la evaluación y no alteran el entorno de llamada o global). El siguiente código lo demuestra:

```
const geval = eval; // Usando otro nombre hace una eval global let x = "global", y = "global"; // Dos variables globales function f() { // Esta función hace una eval local let x = "local"; // Define una variable local eval("x += 'cambiado';"); // La eval directa establece la variable local return x; // Devuelve la variable local cambiada } function g() { // Esta función hace una eval global let y = "local"; // Una variable local geval("y += 'cambiado';"); // La eval indirecta establece la variable global return y; // Devuelve la variable local no cambiada } console.log(f(), x); // Variable local cambiada: imprime "localchanged global": console.log(g(), y); // Variable global cambiada: imprime "local globalchanged":
```

Tenga en cuenta que la capacidad de hacer una evaluación global no es sólo una adaptación a las necesidades del optimizador; en realidad es una característica tremadamente útil que le permite ejecutar cadenas de código como si fueran scripts independientes de alto nivel. Como se señaló al principio de esta sección, es raro que se necesite realmente evaluar una cadena de código. Pero si lo encuentras necesario, es más probable que quieras hacer una eval global que una eval local.

### 4.12.3 Evaluación estricta

El modo estricto (véase §5.6.3) impone más restricciones al comportamiento de la función eval() e incluso al uso del identificador "eval". Cuando se llama a eval() desde código en modo estricto, o cuando la propia cadena de código a evaluar comienza con una directiva "use strict", entonces eval() realiza una evaluación local con un entorno de variables privado. Esto significa que en modo estricto, el código evaluado puede consultar y establecer variables locales, pero no puede definir nuevas variables o funciones en el ámbito local.

Además, el modo estricto hace que eval() sea aún más parecido a un operador, convirtiendo efectivamente a "eval" en una palabra reservada. No se permite sobrescribir la función eval() con un nuevo valor. Y no se permite declarar una variable, función, parámetro de función o parámetro de bloque de captura con el nombre "eval".

## 4.13 Operadores diversos

JavaScript admite otros operadores diversos, que se describen en las siguientes secciones.

### 4.13.1 El operador condicional (?:)

El operador condicional es el único operador ternario (tres operandos) en JavaScript y a veces se le llama *operador ternario*. Este operador a veces se escribe ?:, aunque no aparece así en el código. Como este operador tiene tres operandos, el primero va antes de ?, el segundo va entre ? y :, y el tercero va después de :. Se utiliza así:

$x > 0 ? x : -x$  // El valor absoluto de  $x$

Los operandos del operador condicional pueden ser de cualquier tipo. El primer operando se evalúa y se interpreta como un booleano. Si el valor del primer operando es verdadero, entonces se evalúa el segundo operando y se devuelve su valor. En caso contrario, si el primer operando es falso, entonces se evalúa el tercer operando y se devuelve su valor. Sólo se evalúa uno de los segundos y terceros operandos; nunca ambos.

Aunque puede conseguir resultados similares utilizando la sentencia if ([§5.3.1](#)), el operador ?: suele ser un atajo muy útil. Este es un uso típico, que comprueba que una variable está definida (y tiene un valor significativo y verdadero) y la utiliza si es así o proporciona un valor por defecto si no es así:

```
greeting = "hello " + (username ? username : "there");
```

Esto es equivalente, pero más compacto, a la siguiente sentencia if:

```
greeting = "hello "; if (username) {  
    greeting += username; } else {  
    greeting += "there"; }
```

## 4.13.2 Primera definición (??)

El operador de primera definición ?? evalúa a su primer operando definido: si su operando izquierdo no es nulo ni indefinido, devuelve ese valor. En caso contrario, devuelve el valor del operando derecho. Al igual que los operadores && y ||, el operador ?? es de cortocircuito: sólo evalúa su segundo operando si el primero es nulo

o indefinido. Si la expresión a no tiene efectos secundarios, la expresión a ?? b es equivalente a:

```
(a !== null && a !== undefined) ? a : b
```

El operador || es una alternativa útil a || ([§4.10.2](#)) cuando se quiere seleccionar el primer operando *definido en lugar del primer operando verdadero*. Aunque || es nominalmente un operador lógico OR, también se usa idiomáticamente para seleccionar el primer operando no verdadero con código como este:

```
// Si maxWidth es verdadero, úsallo. Si no, busca un valor en // el objeto de preferencias. Si esto no es cierto, utilice una constante codificada. let max = maxWidth || preferences.maxWidth || 500;
```

El problema de este uso idiomático es que cero, la cadena vacía y false son valores falsos que pueden ser perfectamente válidos en algunas circunstancias. En este ejemplo de código, si maxWidth es cero, ese valor será ignorado. Pero si cambiamos el operador || por ??, terminamos con una expresión en la que el cero es un valor válido:

```
// Si el ancho máximo está definido, utilícelo. Si no, busca un valor en // el objeto de preferencias. Si no está definido, usa una constante codificada. let max = maxWidth ?? preferences.maxWidth ?? 500;
```

Aquí hay más ejemplos que muestran cómo funciona ?? cuando el primer operando es falso. Si ese operando es falso pero está definido, entonces ?? lo devuelve. Sólo cuando el primer operando es "nulo" (es decir, nulo o indefinido) este operador evalúa y devuelve el segundo operando:

```
let options = { timeout: 0, title: "", verbose: false, n: null }; options.timeout ?? 1000  
// => 0: como se define en el objeto options. title ?? "Untitled" // => "": como se
```

*define en las opciones del objeto. verbose ?? true // => false: como se define en las opciones del objeto. quiet ?? false // => false: la propiedad no está definida opciones. n ?? 10 // => 10: la propiedad es nula*

Tenga en cuenta que las expresiones timeout, title y verbose tendrían valores diferentes si utilizáramos || en lugar de ?

El operador ?? es similar a los operadores && y || pero no tiene mayor o menor precedencia que éstos. Si lo usa en una expresión con cualquiera de esos operadores, debe usar paréntesis explícitos para especificar qué operación quiere realizar primero:

(a ?? b) || c // || primero, luego // a ?? (b || c) // //  
primero, luego /?  
a ?? b || c // *SyntaxError: los paréntesis son necesarios*

El operador ?? está definido por ES2020, y a partir de principios de 2020, está recién soportado por las versiones actuales o beta de todos los principales navegadores. Este operador se llama formalmente el operador "coalescente nulo", pero evito ese término porque este operador selecciona uno de sus operandos pero no los "coalesce" de ninguna manera que yo pueda ver.

### 4.13.3 El operador typeof

typeof es un operador unario que se coloca antes de su único operando, que puede ser de cualquier tipo. Su valor es una cadena que especifica el tipo del operando. La Tabla 4-3 especifica el valor del operador typeof para cualquier valor de JavaScript.

*Tabla 4-3. Valores devueltos por el operador typeof*

| x          | tipo de x    |
|------------|--------------|
| indefinido | "indefinido" |

|                               |            |
|-------------------------------|------------|
| null                          | "objeto"   |
| verdadero o falso             | "booleano" |
| cualquier número o NaN        | "número"   |
| cualquier BigInt              | "bigint"   |
| cualquier cadena              | "cadena"   |
| cualquier símbolo             | "símbolo"  |
| cualquier función             | "función"  |
| cualquier objeto no funcional | "objeto"   |

Puede utilizar el operador `typeof` en una expresión como ésta:

```
// Si el valor es una cadena, envuélvalo entre comillas, de lo contrario, convierta
(typeof value === "string") ? "" + valor + "" : valor. toString()
```

Tenga en cuenta que `typeof` devuelve "object" si el valor del operando es `null`. Si quiere distinguir entre `null` y objetos, tendrá que comprobar explícitamente este valor de caso especial.

Aunque las funciones de JavaScript son un tipo de objeto, el operador `typeof` considera que las funciones son lo suficientemente diferentes como para tener su propio valor de retorno.

Dado que `typeof` evalúa a "object" para todos los valores de objetos y arrays que no sean funciones, sólo es útil para distinguir los objetos de otros tipos primitivos. Para distinguir una clase de objeto de otra, debe utilizar otras técnicas, como el operador `instanceof` (véase §4.9.4), el atributo `class` (véase §14.4.3), o la propiedad `constructor` (véase §9.2.2 y §14.3).

#### 4.13.4 El operador de borrado

delete es un operador unario que intenta borrar la propiedad del objeto o el elemento del array especificado como su operando. Al igual que los operadores de asignación, incremento y decremento, delete se utiliza normalmente por su efecto secundario de eliminación de la propiedad y no por el valor que devuelve. Algunos ejemplos:

```
let o = { x: 1, y: 2}; // Empieza con un objeto delete o.x; // Borra una de sus  
propiedades  
"x" en o //=> false: la propiedad ya no existe
```

```
let a = [1,2,3]; // Empezar con un array delete a[2]; // Borrar el último elemento  
del array  
2 en a //=> false: el elemento 2 del array ya no existe  
a.longitud //=> 3: aunque la longitud del array no cambia
```

Tenga en cuenta que una propiedad o elemento de matriz eliminado no se limita a establecer el valor indefinido. Cuando se borra una propiedad, ésta deja de existir. Intentar leer una propiedad inexistente devuelve undefined, pero puede comprobar la existencia real de una propiedad con el operador `in` ([§4.9.3](#)). La eliminación de un elemento de una matriz deja un "agujero" en la matriz y no cambia la longitud de la misma. La matriz resultante es *escasa* ([§7.3](#)).

delete espera que su operando sea un lvalue. Si no es un lvalue, el operador no realiza ninguna acción y devuelve true. En caso contrario, delete intenta eliminar el lvalue especificado. delete devuelve true si elimina con éxito el lvalue especificado. Sin embargo, no todas las propiedades pueden ser eliminadas: las propiedades no configurables ([§14.1](#)) son inmunes a la eliminación.

En modo estricto, delete genera un SyntaxError si su operando es un identificador no cualificado como una variable, función o parámetro de función: sólo funciona cuando el operando es una expresión de acceso a una propiedad ([§4.4](#)). El modo estricto también especifica que delete genera un TypeError si se le pide que elimine cualquier propiedad no configurable (es decir, no eliminable). Fuera del modo estricto, no se produce ninguna excepción en estos casos, y delete simplemente devuelve false para indicar que el operando no pudo ser borrado.

Estos son algunos ejemplos de uso del operador de borrado:

```
let o = {x: 1, y: 2}; delete o.x; // Eliminar una de las propiedades del objeto; devuelve true.  
typeof o.x; // La propiedad no existe; devuelve "undefined". delete o.x; // Borra una propiedad inexistente; devuelve true. delete 1; // Esto no tiene sentido, pero simplemente devuelve true. // No se puede borrar una variable; devuelve false, o SyntaxError en modo estricto. delete o;  
// Propiedad indeleble: devuelve false, o TypeError en modo estricto. delete Object.prototype;
```

Volveremos a ver el operador de borrado en [§6.4](#).

#### 4.13.5 El operador de espera

await se introdujo en ES2017 como una forma de hacer más natural la programación asíncrona en JavaScript. Tendrás que leer [el capítulo 13](#) para entender este operador. Brevemente, sin embargo, await espera un objeto Promise (que representa un cálculo asíncrono) como su único operando, y hace que tu programa se comporte como si estuviera esperando a que el cálculo asíncrono se complete (pero lo hace sin bloquearse realmente, y no impide que otras operaciones asíncronas

procedan al mismo tiempo). El valor del operador await es el valor de cumplimiento del objeto Promise. Es importante destacar que await sólo es legal dentro de funciones que han sido declaradas asíncronas con la palabra clave `async`. Nuevamente, vea [el Capítulo 13](#) para más detalles.

#### 4.13.6 El operador de vacío

`void` es un operador unario que aparece antes de su único operando, que puede ser de cualquier tipo. Este operador es inusual y de uso poco frecuente; evalúa su operando, luego descarta el valor y devuelve `undefined`. Dado que el valor del operando se descarta, el uso del operador `void` sólo tiene sentido si el operando tiene efectos secundarios.

El operador `void` es tan oscuro que es difícil encontrar un ejemplo práctico de su uso. Un caso sería cuando se quiere definir una función que no devuelve nada, pero que también utiliza la sintaxis abreviada de la función flecha (véase [§8.1.3](#)), donde el cuerpo de la función es una única expresión que se evalúa y devuelve. Si se evalúa la expresión únicamente por sus efectos secundarios y no se desea devolver su valor, lo más sencillo es utilizar llaves alrededor del cuerpo de la función.

Pero, como alternativa, también podrías utilizar el operador `void` en este caso:

```
dejar contador = 0;  
const increment = () => void counter++; increment() //  
=> undefined counter // => 1
```

#### 4.13.7 El operador coma (,)

El operador coma es un operador binario cuyos operandos pueden ser de cualquier tipo. Evalúa su operando izquierdo, evalúa su operando derecho y luego devuelve el valor del operando derecho. Así, la siguiente línea:

```
i=0, j=1, k=2;
```

evalúa a 2 y es básicamente equivalente a:

```
i = 0; j = 1; k = 2;
```

La expresión de la izquierda siempre se evalúa, pero su valor se descarta, lo que significa que sólo tiene sentido utilizar el operador coma cuando la expresión de la izquierda tiene efectos secundarios. La única situación en la que el operador coma se utiliza comúnmente es con un bucle for ([§5.4.3](#)) que tiene múltiples variables de bucle:

```
// La primera coma de abajo es parte de la sintaxis de la sentencia let  
// La segunda coma es el operador coma: nos permite apretar  
2  
// expresiones (i++ y j--) en una sentencia (el bucle for) que espera 1.  
for(let i=0,j=10; i < j; i++,j--) { console.log(i+j);  
}
```

## 4.14 Resumen

Este capítulo cubre una amplia variedad de temas, y hay mucho material de referencia aquí que puede querer releer en el futuro mientras continúa aprendiendo JavaScript. Sin embargo, algunos puntos clave que debes recordar son los siguientes:

- Las expresiones son las frases de un programa JavaScript.
- Cualquier expresión puede ser *evaluada* a un valor de JavaScript.

Las expresiones también pueden tener efectos secundarios (como la asignación de variables) además de producir un valor.

- Las expresiones simples, como los literales, las referencias a variables y los accesos a propiedades, pueden combinarse con operadores para producir expresiones más amplias.
- JavaScript define operadores para la aritmética, las comparaciones, la lógica booleana, la asignación y la manipulación de bits, junto con algunos operadores diversos, incluido el operador condicional ternario.
- El operador + de JavaScript se utiliza tanto para sumar números como para concatenar cadenas.
- Los operadores lógicos && y || tienen un comportamiento especial de "cortocircuito" y a veces sólo evalúan uno de sus argumentos. Los modismos comunes de JavaScript requieren que se entienda el comportamiento especial de estos operadores.







## v Capítulo 5. Declaraciones

---

El capítulo 4 describió las expresiones como frases de JavaScript. Por esa analogía, las *sentencias* son frases o comandos de JavaScript. Al igual que las frases en inglés se terminan y se separan unas de otras con puntos, las sentencias en JavaScript se terminan con punto y coma (§2.6). Las expresiones se *evalúan* para producir un valor, pero las sentencias se *ejecutan* para que ocurra algo.

Una forma de "hacer que algo suceda" es evaluar una expresión que tiene efectos secundarios. Las expresiones con efectos secundarios, como las asignaciones y las invocaciones de funciones, pueden funcionar por sí solas como sentencias, y cuando se utilizan de este modo se conocen como sentencias de *expresión*. Una categoría similar de sentencias son las *declaraciones* que declaran nuevas variables y definen nuevas funciones.

Los programas de JavaScript no son más que una secuencia de sentencias a ejecutar. Por defecto, el intérprete de JavaScript ejecuta estas sentencias una tras otra en el orden en que están escritas. Otra forma de "hacer que algo suceda" es alterar este orden de ejecución por defecto, y JavaScript tiene una serie de sentencias o *estructuras de control* que hacen precisamente esto:

### *Condicionales*

Sentencias como if y switch que hacen que el intérprete de JavaScript ejecute u omita otras sentencias en función del valor de una expresión

### *Bucles*

Sentencias como while y for que ejecutan otras sentencias de forma repetitiva

### *Saltos*

Sentencias como break, return y throw que hacen que el intérprete salte a otra parte del programa

Las secciones que siguen describen las distintas sentencias de JavaScript y explican su sintaxis. La [Tabla 5-1](#), al final del capítulo, resume la sintaxis. Un programa de JavaScript es simplemente una secuencia de sentencias, separadas unas de otras con puntos y comas, por lo que una vez que esté familiarizado con las sentencias de JavaScript, podrá empezar a escribir programas de JavaScript.

## **5.1 Declaraciones de expresión**

Los tipos de sentencias más simples en JavaScript son las expresiones que tienen efectos secundarios. Este tipo de expresión se mostró en el [Capítulo 4](#). Las sentencias de asignación son una de las principales categorías de sentencias de expresión. Por ejemplo:

```
saludo = "Hola " + nombre; i *= 3;
```

Los operadores de incremento y decremento, `++` y `--`, están relacionados con las sentencias de asignación. Tienen el efecto secundario de cambiar el valor de una variable, como si se hubiera realizado una asignación:

```
contra++;
```

El operador `delete` tiene el importante efecto secundario de borrar una propiedad del objeto. Por lo tanto, casi siempre se utiliza como una declaración, en lugar de como parte de una expresión mayor:

```
borrar o. x;
```

Las llamadas a funciones son otra categoría importante de declaraciones de expresión. Por ejemplo:

```
console.log(debugMessage); displaySpinner(); // Una función hipotética para  
mostrar un spinner en una aplicación web.
```

Estas llamadas a funciones son expresiones, pero tienen efectos secundarios que afectan al entorno del host o al estado del programa, y se utilizan aquí como sentencias. Si una función no tiene efectos secundarios, no tiene sentido llamarla, a no ser que forme parte de una expresión mayor o de una sentencia de asignación. Por ejemplo, usted no calcularía un coseno y descartaría el resultado:

```
Matemáticas. cos(x);
```

Pero bien podría calcular el valor y asignarlo a una variable para su uso futuro:

```
cx = Math. cos(x);
```

Tenga en cuenta que cada línea de código en cada uno de estos ejemplos termina con un punto y coma.

## 5.2 Declaraciones compuestas y vacías

Al igual que el operador coma ([§4.13.7](#)) combina varias expresiones en una sola, un bloque de *sentencias* combina varias sentencias en una sola *sentencia compuesta*. Un bloque de sentencias es simplemente una secuencia de sentencias encerradas entre llaves. Así, las siguientes líneas actúan como una única sentencia y pueden utilizarse en cualquier lugar en el que JavaScript espere una única sentencia:

```
{ x = Math. PI; cx = Math. cos(x); console.  
log("cos(π) = " + cx); }
```

Hay algunas cosas que hay que tener en cuenta sobre este bloque de declaraciones. En primer lugar, *no* termina con un punto y coma. Las sentencias primitivas dentro del bloque terminan con punto y coma, pero el bloque en sí no lo hace. En segundo lugar, las líneas dentro del bloque tienen una sangría relativa a las llaves que las encierran. Esto es opcional, pero facilita la lectura y comprensión del código.

Al igual que las expresiones contienen a menudo subexpresiones, muchas sentencias de JavaScript contienen subexpresiones. Formalmente, la sintaxis de JavaScript suele permitir una única subexpresión. Por ejemplo, la sintaxis del bucle while incluye una única sentencia que sirve como cuerpo del bucle. Utilizando un bloque

de sentencias, puede colocar cualquier número de sentencias dentro de esta única sustanciación permitida.

Una sentencia compuesta le permite utilizar varias sentencias en las que

La sintaxis de JavaScript espera una única sentencia. La sentencia *vacía* es lo contrario: permite no incluir ninguna sentencia donde se espera una. La sentencia vacía tiene el siguiente aspecto:

```
;
```

El intérprete de JavaScript no realiza ninguna acción cuando ejecuta una sentencia vacía. La sentencia empty es ocasionalmente útil cuando se quiere crear un bucle que tenga el cuerpo vacío.

Considere el siguiente bucle for (los bucles for se tratarán en [§5.4.3](#)):

```
// Inicializar un array a
for(let i = 0; i < a.length; a[i++] = 0);
```

En este bucle, todo el trabajo lo realiza la expresión `a[i++] = 0`, y no es necesario el cuerpo del bucle. Sin embargo, la sintaxis de JavaScript requiere una expresión como cuerpo del bucle, por lo que se utiliza una expresión vacía -sólo un punto y coma-.

Tenga en cuenta que la inclusión accidental de un punto y coma después del paréntesis derecho de un bucle for, un bucle while o una sentencia if puede causar errores frustrantes que son difíciles de detectar. Por ejemplo, el siguiente código probablemente no hace lo que el autor pretendía:

```
if ((a === 0) || (b === 0)); // iOops! Esta línea no hace nada...
o = null; // y esta línea se ejecuta siempre.
```

Cuando utilice intencionadamente la sentencia vacía, es una buena idea comentar su código de forma que quede claro que lo hace a propósito. Por ejemplo:

```
for(let i = 0; i < a. length; a[i++] = 0) /* empty */;
```

## 5.3 Condicionales

Las sentencias condicionales ejecutan o saltan otras sentencias dependiendo del valor de una expresión especificada. Estas sentencias son los puntos de decisión de tu código, y también se conocen a veces como "ramas". Si imagina que un intérprete de JavaScript sigue un camino a través de su código, las sentencias condicionales son los lugares donde el código se bifurca en dos o más caminos y el intérprete debe elegir qué camino seguir.

Las siguientes subsecciones explican la condicional básica de JavaScript, la sentencia if/else, y también cubren el switch, una sentencia de bifurcación más complicada.

### 5.3.1 si

La sentencia if es la sentencia de control fundamental que permite JavaScript para tomar decisiones o, más precisamente, para ejecutar declaraciones condicionalmente. Esta declaración tiene dos formas. La primera es:

*Declaración if (expresión)*

En esta forma, se evalúa la expresión. Si el valor resultante es verdadero, se ejecuta la *expresión*. Si la *expresión* es falsa, la *sentencia* no se ejecuta.

(Véase en [§3.4](#) la definición de los valores de verdad y de mentira):

```
if (username == null) // Si el nombre de usuario es nulo o no está definido,  
username = "John Doe"; // defínalo
```

O de forma similar:

```
// Si el nombre de usuario es nulo, indefinido, falso, 0, "", o NaN, dale un nuevo valor  
if (! nombre de usuario) nombre de usuario = "John Doe";
```

Tenga en cuenta que los paréntesis alrededor de la *expresión* son una parte obligatoria de la sintaxis de la sentencia if.

La sintaxis de JavaScript requiere una única sentencia después de la palabra clave if y la expresión entre paréntesis, pero puede utilizar un bloque de sentencias para combinar varias sentencias en una sola. Así que la sentencia if también podría tener este aspecto:

```
if (! dirección) { dirección = ""; mensaje = "Por favor, especifique una  
dirección postal."; }
```

La segunda forma de la sentencia if introduce una cláusula else que se ejecuta cuando *la expresión* es falsa. Su sintaxis es:

```
if (expresión)  
  statement1 else  
  statement2
```

Esta forma de la sentencia ejecuta la sentencia1 si *la expresión* es verdadera y ejecuta la sentencia2 si *la expresión* es falsa. Por ejemplo:

```
si (n === 1)  
  console. log("Tiene 1 mensaje nuevo."); else  
  console. log(`Tienes ${n} nuevos mensajes.');
```

Cuando se tienen sentencias if anidadas con cláusulas else, es necesario tener cierta precaución para asegurarse de que la cláusula else va con la sentencia if apropiada. Considere las siguientes líneas:

```
i = j = 1; k = 2; if (i ===  
j) if (j === k)  
    console.log("i es igual a k"); else console.log("i no es igual a j"); //  
    ijjError!!!
```

En este ejemplo, la sentencia if interna forma la única sentencia permitida por la sintaxis de la sentencia if externa.

Desgraciadamente, no está claro (salvo por la pista que da la sangría) con qué if va el else. Y en este ejemplo, la sangría es incorrecta, porque un intérprete de JavaScript interpreta realmente el ejemplo anterior como:

```
si (i === j) { si (j === k)  
    console.log("i es igual a k"); else console.log("i no es igual a j"); //  
    iOOPS!  
}
```

La regla en JavaScript (como en la mayoría de los lenguajes de programación) es que, por defecto, una cláusula else forma parte de la sentencia if más cercana. Para que este ejemplo sea menos ambiguo y más fácil de leer, entender, mantener y depurar, debe utilizar llaves:

```
if (i === j) { if (j === k) { console.log("i es igual a  
k"); }  
} else { // iQué diferencia hace la ubicación de una llave rizada!  
    console.log("i no es igual a j");  
}
```

Muchos programadores tienen la costumbre de encerrar los cuerpos de las sentencias if y else (así como otras sentencias compuestas, como los bucles while) entre llaves, incluso cuando el cuerpo consiste en una sola sentencia. Hacerlo de forma consistente puede evitar el tipo de problema que se acaba de mostrar, y le aconsejo que adopte esta práctica. En este libro impreso, doy mucha importancia a mantener el código de ejemplo verticalmente compacto, y no siempre sigo mi propio consejo en este asunto.

### 5.3.2 si no

La sentencia if/else evalúa una expresión y ejecuta uno de los dos fragmentos de código, dependiendo del resultado. Pero, ¿qué ocurre cuando se necesita ejecutar una de las muchas partes del código? Una forma de hacerlo es con una sentencia else if. else if no es realmente una sentencia de JavaScript, sino simplemente un lenguaje de programación de uso frecuente que resulta cuando se utilizan sentencias if/else repetidas:

```

if (n === 1) { // Ejecutar el bloque de
código #1
} else if (n === 2) {
    // Ejecutar el bloque de código #2
} else if (n === 3) {
    // Ejecutar el bloque de código #3
} si no {
    // Si todo lo demás falla, ejecuta el bloque #4
}

```

Este código no tiene nada de especial. Es simplemente una serie de sentencias if, donde cada if siguiente es parte de la cláusula else de la sentencia anterior. Usar el lenguaje else if es preferible y más legible que escribir estas sentencias en su forma sintáctica equivalente, completamente anidada:

```

if (n === 1) { // Ejecuta el bloque de código #1
} else { if (n === 2) { // Ejecuta el bloque de
código #2 } else {
    if (n === 3) { // Ejecuta el bloque de código #3 } else { // Si todo lo
demás falla, ejecuta el bloque #4
    }
}
}

```

### 5.3.3 interruptor

Una sentencia if provoca una bifurcación en el flujo de ejecución de un programa, y se puede utilizar el lenguaje else if para realizar una bifurcación multidireccional. Sin embargo, esta no es la mejor

solución cuando todas las ramas dependen del valor de la misma expresión. En este caso, es un desperdicio evaluar repetidamente esa expresión en múltiples sentencias if.

La sentencia switch maneja exactamente esta situación. La palabra clave switch va seguida de una expresión entre paréntesis y un bloque de código entre llaves:

```
switch(expresión) {  
declaraciones }
```

Sin embargo, la sintaxis completa de una sentencia switch es más compleja que esto. Varias ubicaciones en el bloque de código se etiquetan con la palabra clave case seguida de una expresión y dos puntos. Cuando un switch se ejecuta, calcula el valor de la *expresión* y luego busca una etiqueta case cuya expresión se evalúe con el mismo valor (donde la igualdad está determinada por el operador `==`). Si encuentra uno, comienza a ejecutar el bloque de código en la declaración etiquetada por el caso. Si no encuentra un caso con un valor que coincida, busca una sentencia etiquetada como default:. Si no hay ninguna etiqueta default:, la sentencia switch se salta el bloque de código.

La sentencia switch es una sentencia confusa de explicar; su funcionamiento queda mucho más claro con un ejemplo. La siguiente sentencia switch es equivalente a las sentencias repetidas if/else mostradas en la sección anterior:

```
switch(n) { case 1: // Empieza aquí si n === 1 // Ejecuta el bloque de código #1.
```

**break;** // Parar aquí **caso 2:** // Empezar aquí si  $n == 2$  // Ejecutar el bloque de código #2.

**break;** // Parar aquí **caso 3:** // Empezar aquí si  $n == 3$  // Ejecutar el bloque de código #3.

**break;** // Deténgase aquí

```
por defecto:           // Si todo lo demás falla... // Ejecuta el
bloque de código #4.
break; // Deténgase aquí }
```

Observe la palabra clave `break` utilizada al final de cada caso en este código. La sentencia `break`, descrita más adelante en este capítulo, hace que el intérprete salte al final (o "rompa") de la sentencia `switch` y continúe con la sentencia que le sigue. Las cláusulas `case` en una sentencia `switch` especifican sólo el *punto de inicio del código deseado*; no especifican ningún punto final. En ausencia de sentencias `break`, una sentencia `switch` comienza a ejecutar su bloque de código en la etiqueta `case` que coincide con el valor de su *expresión* y continúa ejecutando sentencias hasta que llega al final del bloque. En raras ocasiones, es útil escribir código como este que "pasa" de una etiqueta de caso a la siguiente, pero el 99% de las veces debe tener cuidado de terminar cada caso con una sentencia `break`. (Sin embargo, cuando se utiliza un `switch` dentro de una función, puede utilizar una sentencia `return` en lugar de una sentencia `break`. Ambas sirven para terminar la sentencia `switch` y evitar que la ejecución pase al siguiente caso).

Este es un ejemplo más realista de la sentencia `switch`; convierte un valor en una cadena de manera que depende del tipo del valor:

```
function convert(x) { switch(typeof x) { case "number":           // Convierte el
número en un entero hexadecimal return x.toString(16); case "string":      //
Devuelve la cadena encerrada entre comillas
devolver '"" + x + ""; por defecto:           // Convierte cualquier otro tipo de
la forma habitual return String(x); }
}
```

Observe que en los dos ejemplos anteriores, las palabras clave case van seguidas de literales de número y de cadena, respectivamente. Esta es la forma en que la sentencia switch se utiliza más a menudo en la práctica, pero tenga en cuenta que el estándar ECMAScript permite que cada caso sea seguido por una expresión arbitraria.

La sentencia switch evalúa primero la expresión que sigue a la palabra clave switch y luego evalúa las expresiones case, en el orden en que aparecen, hasta encontrar un valor que coincida.<sup>1</sup> El caso coincidente se determina utilizando el operador de identidad ===, no el operador de igualdad ==, por lo que las expresiones deben coincidir sin ninguna conversión de tipo.

Dado que no todas las expresiones case se evalúan cada vez que se ejecuta la sentencia switch, debe evitar utilizar expresiones case que contengan efectos secundarios como llamadas a funciones o asignaciones. Lo más seguro es limitar las expresiones case a expresiones constantes.

Como se ha explicado anteriormente, si ninguna de las expresiones case coincide con la expresión switch, la sentencia switch comienza a ejecutar su cuerpo en la sentencia etiquetada como default:. Si no hay ninguna etiqueta default:, la sentencia switch se salta su cuerpo por completo. Observe que en los ejemplos mostrados, la etiqueta default: aparece al final del cuerpo de la sentencia switch, después de todas las etiquetas case. Este es un lugar lógico y común para ella, pero en realidad puede aparecer en cualquier lugar dentro del cuerpo de la sentencia.

## 5.4 Bucles

Para entender las sentencias condicionales, nos imaginamos al intérprete de JavaScript siguiendo un camino de bifurcación a través de tu código fuente. Las sentencias de *bucle* son las que doblan ese camino sobre sí mismo para repetir partes de su código. JavaScript tiene cinco sentencias de bucle: while, do/while, for, for/of (y su variante for/await), y for/in. Las siguientes subsecciones explican cada una de ellas. Un uso común de los bucles es iterar sobre los elementos de un array. En §7.6 se discute este tipo de bucle en detalle y se cubren los métodos especiales de bucle definidos por la clase Array.

### 5.4.1 mientras

Al igual que la sentencia if es el condicional básico de JavaScript, la sentencia while es el bucle básico de JavaScript. Tiene la siguiente sintaxis:

*declaración while  
(expresión)*

Para ejecutar una sentencia while, el intérprete evalúa primero *la expresión*. Si el valor de la expresión es falso, el intérprete se salta la *sentencia* que sirve de cuerpo del bucle y pasa a la siguiente sentencia del programa. Si, por el contrario, la *expresión es verdadera*, el intérprete ejecuta la *sentencia* y repite, saltando de nuevo al principio del bucle y evaluando *la expresión* de nuevo. Otra forma de decir esto es que el intérprete ejecuta *la sentencia* repetidamente *mientras la expresión es verdadera*. Observe que puede crear un bucle infinito con la sintaxis while(true).

Normalmente, no se quiere que JavaScript realice exactamente la misma operación una y otra vez. En casi todos los bucles, una o más variables cambian en cada *iteración* del bucle. Dado que las variables cambian, las acciones realizadas por la ejecución de *la expresión* pueden diferir cada vez que se pasa por el bucle.

Además, si la variable o variables que cambian están involucradas en una *expresión*, el valor de la expresión puede ser diferente cada vez que se ejecuta el bucle. Esto es importante; de lo contrario, una expresión que comienza siendo verdadera nunca cambiaría, y el bucle nunca terminaría! Aquí hay un ejemplo de un bucle while que imprime los números del 0 al 9:

```
let count = 0; while(count <
10) { console.log(count);
count++; }
```

Como puede ver, la cuenta de la variable comienza en 0 y se incrementa cada vez que se ejecuta el cuerpo del bucle. Una vez que el bucle se ha ejecutado 10 veces, la expresión se convierte en falsa (es decir, la cuenta de la variable ya no es menor de 10), la sentencia while termina, y el intérprete puede pasar a la siguiente sentencia del programa. Muchos bucles tienen una variable contadora como count. Los nombres de las variables i, j y k se usan comúnmente como contadores de bucles, aunque debería usar nombres más descriptivos si hace que su código sea más fácil de entender.

## 5.4.2 hacer/mientras

El bucle do/while es como un bucle while, excepto que la expresión del bucle se comprueba al final del bucle en lugar de al

principio. Esto significa que el cuerpo del bucle siempre se ejecuta al menos una vez. La sintaxis es:

*declaración do while  
(expresión);*

El bucle do/while es menos utilizado que su primo while- en la práctica, es poco común estar seguro de querer que un bucle se ejecute al menos una vez. Aquí hay un ejemplo de un bucle do/while:

```
function printArray(a) { let len = a.length, i = 0; if (len === 0) { console.log("Empty Array"); } else { do { console.log(a[i]); } while(++i < len); }
```

Hay un par de diferencias sintácticas entre el bucle do/while y el bucle while ordinario. En primer lugar, el bucle do requiere tanto la palabra clave do (para marcar el inicio del bucle) como la palabra clave while (para marcar el final e introducir la condición del bucle). Además, el bucle do debe terminar siempre con un punto y coma. El bucle while no necesita un punto y coma si el cuerpo del bucle está encerrado entre llaves.

### 5.4.3 para

La sentencia for proporciona una construcción de bucle que a menudo es más conveniente que la sentencia while. La sentencia for simplifica los bucles que siguen un patrón común. La mayoría de los bucles tienen una variable contadora de algún tipo. Esta variable se inicializa antes de que comience el bucle y se comprueba antes de cada iteración del bucle. Finalmente, la variable contador se incrementa o se actualiza de alguna manera al

final del cuerpo del bucle, justo antes de que la variable sea probada de nuevo. En este tipo de bucle, la inicialización, la prueba y la actualización son las tres manipulaciones cruciales de una variable de bucle. La sentencia `for` codifica cada una de estas tres manipulaciones como una expresión y hace que esas expresiones sean parte explícita de la sintaxis del bucle:

*Declaración for(initialize ; test ; increment)*

*inicializar*, *probar* e *incrementar* son tres expresiones (separadas por punto y coma) que son responsables de inicializar, probar e incrementar la variable del bucle. Ponerlas todas en la primera línea del bucle facilita la comprensión de lo que hace un bucle `for` y evita errores como olvidar inicializar o incrementar la variable del bucle.

La forma más sencilla de explicar cómo funciona un bucle `for` es mostrar el bucle `while` equivalente:<sup>2</sup>

```
initialize; while(test) { statement increment;  
}
```

En otras palabras, la expresión de *inicialización* se evalúa una vez, antes de que comience el bucle. Para ser útil, esta expresión debe tener efectos secundarios (normalmente una asignación).

JavaScript también permite que *initialize* sea una sentencia de declaración de variables, de modo que se puede declarar e inicializar un contador de bucle al mismo tiempo. La expresión *test* se evalúa antes de cada iteración y controla si se ejecuta el cuerpo del bucle. Si *test* evalúa un valor verdadero, se ejecuta *la sentencia* que es el cuerpo del bucle. Por último, se evalúa la expresión de *incremento*. De nuevo, debe ser una expresión con efectos

secundarios para que sea útil. Generalmente, es una expresión de asignación, o utiliza los operadores ++ o -.

Podemos imprimir los números del 0 al 9 con un bucle for como el siguiente. Contrastá con el bucle while equivalente mostrado en la sección anterior:

```
for(let count = 0; count < 10; count++) { console.  
log(count); }
```

Los bucles pueden ser mucho más complejos que este simple ejemplo, por supuesto, y a veces múltiples variables cambian con cada iteración del bucle. Esta situación es el único lugar en el que el operador coma se utiliza comúnmente en JavaScript; proporciona una manera de combinar múltiples expresiones de inicialización e incremento en una sola expresión adecuada para su uso en un bucle for:

```
let i, j, sum = 0; for(i = 0, j = 10 ; i < 10 ; i++, j--) { sum += i * j;  
}
```

En todos nuestros ejemplos de bucles hasta ahora, la variable del bucle ha sido numérica. Esto es bastante común pero no es necesario. El siguiente código utiliza un bucle for para recorrer una estructura de datos de lista enlazada y devolver el último objeto de la lista (es decir, el primer objeto que no tiene una propiedad next):

```
function tail(o) { // Devuelve la cola de la lista  
enlazada o for(; o.next; o = o.next) /* empty */;  
// Recorre mientras  
o.next es verdadero return o; }
```

Observe que este código no tiene ninguna expresión de *inicialización*. Cualquiera de las tres expresiones puede ser omitida en un bucle `for`, pero los dos puntos y coma son necesarios. Si omite la expresión de *prueba*, el bucle se repite para siempre, y `for(;;)` es otra forma de escribir un bucle infinito, como `while(true)`.

#### 5.4.4 para/de

ES6 define una nueva sentencia de bucle: `for/of`. Este nuevo tipo de bucle utiliza la palabra clave `for` pero es un tipo de bucle completamente diferente al bucle `for` normal. (También es completamente diferente del antiguo bucle `for/in` que describiremos en §5.4.5.)

El bucle `for/of` funciona con objetos iterables. Explicaremos exactamente lo que significa que un objeto sea iterable en [el capítulo 12](#), pero para este capítulo, es suficiente saber que los arrays, las cadenas, los conjuntos y los mapas son iterables: representan una secuencia o conjunto de elementos que puedes recorrer en bucle o iterar mediante un bucle `for/of`.

Aquí, por ejemplo, es cómo podemos utilizar `for/of` para recorrer los elementos de una matriz de números y calcular su suma:

```
let data = [1, 2, 3, 4, 5, 6, 7, 8, 9], sum = 0; for(let element of data) {  
    sum += element;  
} suma //=> 45
```

Superficialmente, la sintaxis se parece a la de un bucle `for` normal: la palabra clave `for` va seguida de paréntesis que contienen detalles sobre lo que debe hacer el bucle. En este caso, los paréntesis contienen una declaración de variable (`o`, para las

variables que ya han sido declaradas, simplemente el nombre de la variable) seguida de la palabra clave of y una expresión que se evalúa a un objeto iterable, como el array de datos en este caso. Como con todos los bucles, el cuerpo de un bucle for/of sigue a los paréntesis, normalmente entre llaves.

En el código que se acaba de mostrar, el cuerpo del bucle se ejecuta una vez por cada elemento del array de datos. Antes de cada ejecución del cuerpo del bucle, el siguiente elemento del array se asigna a la variable elemento. Los elementos del array se iteran en orden del primero al último.

Los arrays se iteran "en vivo"-los cambios realizados durante la iteración pueden afectar al resultado de la misma. Si modificamos el código anterior añadiendo la línea data.push(sum); dentro del cuerpo del bucle, entonces creamos un bucle infinito porque la iteración nunca puede alcanzar el último elemento del array.

## PARA/DE CON OBJETOS

Los objetos no son (por defecto) iterables. El intento de utilizar for/of en un objeto normal arroja un TypeError en tiempo de ejecución:

```
let o = { x: 1, y: 2, z: 3 }; for(let element of o) { // Lanza TypeError porque o no es iterable
  console.log(element);
}
```

Si quieras iterar a través de las propiedades de un objeto, puedes utilizar el bucle for/in (introducido en §5.4.5), o utilizar for/of con el método Object.keys():

```
let o = { x: 1, y: 2, z: 3 }; let keys = ""; for(let k of Object.keys(o)) { keys += k; } keys //=> "xyz"
```

Esto funciona porque `Object.keys()` devuelve un array de nombres de propiedades de un objeto, y los arrays son iterables con `for/of`. Tenga en cuenta también que esta iteración de las claves de un objeto no es en vivo como el ejemplo de la matriz anterior - los cambios en el objeto o realizados en el cuerpo del bucle no tendrán efecto en la iteración. Si no te importan las claves de un objeto, también puedes iterar a través de sus valores correspondientes así:

```
let sum = 0; for(let v of Object.values(o)) { sum += v; } suma //=> 6
```

Y si le interesan tanto las claves como los valores de las propiedades de un objeto, puede utilizar `for/of` con `Object.entries()` y desestructurar la asignación:

```
let pairs = ""; for(let [k, v] of Object.entries(o)) { pairs += k + v; } pares //=> "x1y2z3"
```

`Object.entries()` devuelve un array de arrays, donde cada array interior representa un par clave/valor para una propiedad del objeto. En este ejemplo de código utilizamos la asignación de desestructuración para descomponer esas matrices internas en dos variables individuales.

## PARA/DE CON CADENAS

Las cadenas son iterables carácter por carácter en ES6:

```
let frequency = {};
for(let letter of "mississippi") {
  if (frequency[letter]) { frequency[letter]++; } else
  { frequency[letter] = 1; }
} frecuencia //=> {m: 1, i: 4, s: 4, p: 2}
```

Tenga en cuenta que las cadenas se iteran por punto de código Unicode, no por carácter UTF-16. La cadena "I ❤️ 🐕" tiene una longitud de 5 (porque los dos caracteres emoji requieren cada uno dos caracteres UTF-16 para ser representados). Pero si iteras esa cadena con for/of, el cuerpo del bucle se ejecutará tres veces, una por cada uno de los tres puntos de código "I", "❤️" y "🐕."

## FOR/OF CON SET Y MAP

Las clases Set y Map incorporadas en ES6 son iterables. Cuando se itera un

Si se establece con for/of, el cuerpo del bucle se ejecuta una vez por cada elemento del conjunto. Se podría utilizar un código como éste para imprimir las palabras únicas de una cadena de texto:

```
let text = "¡Na na na na na Batman!";
let wordSet = new Set(text.split(" "));
let unique = [];
for(let word of wordSet) {
  unique.push(word);
}
unique //=> ["Na", "na", "Batman!"]
```

Los mapas son un caso interesante porque el iterador de un objeto Map no itera las claves de Map, ni los valores de Map, sino pares clave/valor. Cada vez que se itera, el iterador devuelve un array

cuyo primer elemento es una clave y cuyo segundo elemento es el valor correspondiente. Dado un Mapa m, se podría iterar y desestructurar sus pares clave/valor de la siguiente manera:

```
let m = new Map([[1, "uno"]]); for(let  
[clave, valor] of m) { clave //=> 1 valor  
// => "uno" }
```

## ITERACIÓN ASÍNCRONA CON FOR/AWAIT

ES2018 introduce un nuevo tipo de iterador, conocido como *iterador asíncrono*, y una variante del bucle for/of, conocida como bucle for/await que funciona con iteradores asíncronos.

Tendrás que leer los capítulos [12](#) y [13 para](#) entender el bucle for/await, pero aquí tienes su aspecto en código:

```
// Leer trozos de un flujo iterable asíncrono e imprimirlas async function  
printStream(stream) { for await (let chunk of stream) { console.log(chunk); } }
```

## 5.4.5 para/in

Un bucle for/in se parece mucho a un bucle for/of, con la palabra clave of cambiada por in. Mientras que un bucle for/of requiere un objeto iterable después de of, un bucle for/in funciona con cualquier objeto después de in. El bucle for/of es nuevo en ES6, pero for/in ha formado parte de JavaScript desde el principio (por eso tiene una sintaxis más natural).

La sentencia for/in recorre los nombres de las propiedades de un objeto especificado. La sintaxis es la siguiente:

```
for (variable in object) declaración
```

*variable* suele nombrar una variable, pero puede ser una declaración de variable o cualquier cosa adecuada como lado izquierdo de una expresión de asignación. *object* es una expresión que se evalúa a un objeto. Como es habitual, *statement* es la declaración o bloque de declaraciones que sirve como cuerpo del bucle.

Y podrías usar un bucle for/in como este:

```
for(let p in o) { // Asigna los nombres de las propiedades de o a la variable p  
  console.log(o[p]); // Imprime el valor de cada propiedad }
```

Para ejecutar una sentencia for/in, el intérprete de JavaScript evalúa primero la expresión *del objeto*. Si es nula o indefinida, el intérprete omite el bucle y pasa a la siguiente sentencia. El intérprete ejecuta ahora el cuerpo del bucle una vez por cada propiedad enumerable del objeto. Sin embargo, antes de cada iteración, el intérprete evalúa la expresión de la *variable* y le asigna el nombre de la propiedad (un valor de cadena).

Tenga en cuenta que la *variable* en el bucle for/in puede ser una expresión arbitraria, siempre que se evalúe como algo adecuado para el lado izquierdo de una asignación. Esta expresión se evalúa cada vez que se pasa por el bucle, lo que significa que puede evaluarse de forma diferente cada vez. Por ejemplo, puede utilizar un código como el siguiente para copiar los nombres de todas las propiedades del objeto en un array:

```
let o = { x: 1, y: 2, z: 3 }; let a = [], i =  
0; for(a[i++] in o) /* empty */;
```

Los arrays de JavaScript son simplemente un tipo especializado de objeto, y los índices de los arrays son propiedades del objeto que pueden ser enumeradas con un bucle for/in. Por ejemplo, siguiendo el código anterior con esta línea se enumeran los índices del array 0, 1 y 2:

```
for(let i in a) console.log(i);
```

Una fuente común de errores en mi propio código es el uso accidental de for/in con arrays cuando quería usar for/of. Cuando se trabaja con arrays, casi siempre se quiere usar for/of en lugar de for/in.

El bucle for/in no enumera realmente todas las propiedades de un objeto. No enumera las propiedades cuyos nombres son símbolos. Y de las propiedades cuyos nombres son cadenas, sólo hace un bucle sobre las propiedades *enumerables* (véase §14.1). Los diversos métodos incorporados definidos por el núcleo de JavaScript no son enumerables. Todos los objetos tienen un método `toString()`, por ejemplo, pero el bucle for/in no enumera esta propiedad `toString`. Además de los métodos incorporados, muchas otras propiedades de los objetos incorporados no son enumerables. Todas las propiedades y métodos definidos por su código son enumerables, por defecto. (Puede hacerlos no enumerables utilizando las técnicas explicadas en §14.1.)

Las propiedades heredadas enumerables (véase §6.3.2) también son enumeradas por el bucle for/in. Esto significa que si usted utiliza bucles for/in y también utiliza código que define propiedades que son heredadas por todos los objetos, entonces su bucle puede no comportarse de la manera que usted espera. Por

esta razón, muchos programadores prefieren utilizar un bucle for/of con Object.keys() en lugar de un bucle for/in.

Si el cuerpo de un bucle for/in borra una propiedad que aún no ha sido enumerada, esa propiedad no será enumerada. Si el cuerpo del bucle define nuevas propiedades en el objeto, esas propiedades pueden o no ser enumeradas. Véase §6.6.1 para más información sobre el orden en que for/in enumera las propiedades de un objeto.

## 5,5 Saltos

Otra categoría de sentencias de JavaScript son las *sentencias de salto*. Como su nombre indica, hacen que el intérprete de JavaScript salte a una nueva ubicación en el código fuente. La sentencia break hace que el intérprete salte al final de un bucle o de otra sentencia. continue hace que el intérprete se salte el resto del cuerpo de un bucle y vuelva a saltar al principio de un bucle para empezar una nueva iteración. JavaScript permite nombrar las sentencias, o *etiquetarlas*, y break y continue pueden identificar la etiqueta del bucle de destino o de otra sentencia.

La sentencia return hace que el intérprete salte desde la invocación de una función de vuelta al código que la invocó y también proporciona el valor de la invocación. La sentencia throw es una especie de retorno intermedio de una función generadora. La sentencia throw *lanza* una excepción y está diseñada para trabajar con la sentencia try/catch/finally, que establece un bloque de código de gestión de excepciones. Se trata de una complicada sentencia de salto: cuando se lanza una excepción, el intérprete

salta al manejador de excepciones más cercano, que puede estar en la misma función o en la pila de llamadas de una función invocadora.

Los detalles sobre cada una de estas declaraciones de salto se encuentran en las secciones siguientes.

### 5.5.1 Declaraciones etiquetadas

Cualquier declaración puede ser *etiquetada* precediéndola de un identificador y dos puntos:

*identificador: declaración*

Al etiquetar una sentencia, le das un nombre que puedes usar para referirte a ella en cualquier parte de tu programa. Puede etiquetar cualquier sentencia, aunque sólo es útil para etiquetar sentencias que tienen cuerpo, como los bucles y las condicionales. Al dar un nombre a un bucle, puede utilizar las sentencias break y continue dentro del cuerpo del bucle para salir de él o para saltar directamente al principio del bucle para comenzar la siguiente iteración. break y continue son las únicas sentencias de JavaScript que utilizan etiquetas de sentencia; se tratan en las siguientes subsecciones. Este es un ejemplo de un bucle while etiquetado y una sentencia continue que utiliza la etiqueta.

```
mainloop: while(token !== null) { // Código omitido... continue mainloop; // Salta a
  la siguiente iteración del bucle con nombre // Más código omitido...
}
```

El *identificador* que se utiliza para etiquetar una sentencia puede ser cualquier identificador legal de JavaScript que no sea una palabra reservada. El espacio de nombres para las etiquetas es

diferente del espacio de nombres para las variables y las funciones, por lo que se puede utilizar el mismo identificador como etiqueta de sentencia y como nombre de variable o función. Las etiquetas de las sentencias sólo se definen dentro de la sentencia a la que se aplican (y dentro de sus substitutos, por supuesto). Una sentencia no puede tener la misma etiqueta que la sentencia que la contiene, pero dos sentencias pueden tener la misma etiqueta siempre que ninguna esté anidada dentro de la otra. Las sentencias etiquetadas pueden ser a su vez etiquetadas. En efecto, esto significa que cualquier sentencia puede tener varias etiquetas.

## 5.5.2 ruptura

La sentencia break, utilizada por sí sola, hace que el bucle o la sentencia switch más internos salgan inmediatamente. Su sintaxis es sencilla:

**romper;**

Dado que provoca la salida de un bucle o de un switch, esta forma de la sentencia break sólo es legal si aparece dentro de una de estas sentencias.

Ya ha visto ejemplos de la sentencia break dentro de una sentencia switch. En los bucles, se utiliza normalmente para salir prematuramente cuando, por cualquier razón, ya no es necesario completar el bucle. Cuando un bucle tiene condiciones de terminación complejas, a menudo es más fácil implementar algunas de estas condiciones con sentencias break en lugar de intentar expresarlas todas en una única expresión de bucle. El siguiente código busca un valor determinado en los elementos de

un array. El bucle termina de forma normal cuando llega al final del array; termina con una sentencia break si encuentra lo que busca en el array:

```
for(let i = 0; i < a.length; i++) { if (a[i] ===  
target) break; }
```

JavaScript también permite que la palabra clave break vaya seguida de una etiqueta de declaración (sólo el identificador, sin dos puntos):

```
romper labelname;
```

Cuando se utiliza break con una etiqueta, salta al final o termina la sentencia que la encierra y que tiene la etiqueta especificada. Es un error de sintaxis utilizar break de esta forma si no hay una sentencia que la encierre con la etiqueta especificada. Con esta forma de la sentencia break, no es necesario que la sentencia nombrada sea un bucle o un switch: break puede "salir" de cualquier sentencia adjunta. Esta sentencia puede ser incluso un bloque de sentencias agrupado entre llaves con el único propósito de nombrar el bloque con una etiqueta.

No se permite una nueva línea entre la palabra clave break y el *nombre de la etiqueta*. Esto es el resultado de la inserción automática de JavaScript de los puntos y comas omitidos: si pone un terminador de línea entre la palabra clave break y la etiqueta que le sigue, JavaScript asume que quiso usar la forma simple y sin etiqueta de la declaración y trata el terminador de línea como un punto y coma. (Véase §2.6.)

Necesita la forma etiquetada de la sentencia break cuando quiera salir de una sentencia que no sea el bucle más cercano o un switch. El siguiente código lo demuestra:

```
let matrix = getData(); // Obtener una matriz 2D de números de algún lugar //
Ahora sumar todos los números de la matriz. let sum = 0, success = false;
// Comienza con una sentencia etiquetada de la que podemos salir si se producen errores computeSum: if (matrix) { for(let x = 0; x < matrix.length; x++) { let row =
matrix[x]; if (!row) break computeSum;

for(let y = 0; y < row.length; y++) { let cell = row[y]; if
(isNaN(cell)) break computeSum; sum += cell; } } success = true;
}

// Aquí saltan las sentencias break. Si llegamos aquí con success == false //
entonces había algo mal en la matriz que nos dieron. // En caso contrario, sum
contiene la suma de todas las celdas de la matriz.
```

Por último, tenga en cuenta que una sentencia break, con o sin etiqueta, no puede transferir el control a través de los límites de la función. No puede etiquetar una sentencia de definición de función, por ejemplo, y luego utilizar esa etiqueta dentro de la función.

### 5.5.3 continuar

La sentencia continue es similar a la sentencia break. Sin embargo, en lugar de salir de un bucle, continue reinicia un bucle en la

siguiente iteración. La sintaxis de la sentencia continue es tan sencilla como la de la sentencia break:

**continuar;**

La sentencia continue también puede utilizarse con una etiqueta:

continuar con *el nombre de la etiqueta*;

La sentencia continue, tanto en su forma etiquetada como no etiquetada, sólo puede utilizarse dentro del cuerpo de un bucle. Su uso en cualquier otro lugar provoca un error de sintaxis.

Cuando se ejecuta la sentencia continue, la iteración actual del bucle que lo rodea termina y comienza la siguiente iteración. Esto significa diferentes cosas para diferentes tipos de bucles:

- En un bucle while, la *expresión* especificada al principio del bucle se comprueba de nuevo, y si es verdadera, el cuerpo del bucle se ejecuta empezando por el principio.
- En un bucle do/while, la ejecución salta a la parte inferior del bucle, donde la condición del bucle se comprueba de nuevo antes de reiniciar el bucle en la parte superior.
- En un bucle for, se evalúa la expresión de *incremento* y se vuelve a probar la expresión de *prueba* para determinar si se debe hacer otra iteración.
- En un bucle for/of o for/in, el bucle comienza de nuevo con el siguiente valor iterado o el siguiente nombre de propiedad que se asigna a la variable especificada.

Observe la diferencia en el comportamiento de la sentencia continue en los bucles while y for: un bucle while vuelve

directamente a su condición, pero un bucle for evalúa primero su expresión de *incremento* y luego vuelve a su condición.

Anteriormente, consideramos el comportamiento del bucle for en términos de un bucle while "equivalente". Sin embargo, debido a que la sentencia continue se comporta de forma diferente para estos dos bucles, en realidad no es posible simular perfectamente un bucle for con un bucle while solo.

El siguiente ejemplo muestra una sentencia continue no etiquetada que se utiliza para saltar el resto de la iteración actual de un bucle cuando se produce un error:

```
for(let i = 0; i < data.length; i++) { if (!data[i]) continue; // No se puede proceder  
con datos indefinidos total += data[i]; }
```

Al igual que la sentencia break, la sentencia continue se puede utilizar en su forma etiquetada dentro de bucles anidados cuando el bucle que se va a reiniciar no es el bucle inmediatamente contiguo. Además, al igual que con la sentencia break, no se permiten saltos de línea entre la sentencia continue y su *nombre de etiqueta*.

#### 5.5.4 retorno

Recordemos que las invocaciones a funciones son expresiones y que todas las expresiones tienen valores. Una sentencia return dentro de una función especifica el valor de las invocaciones de esa función. Esta es la sintaxis de la sentencia return:

*expresión de retorno;*

Una sentencia return sólo puede aparecer dentro del cuerpo de una función. Es un error sintáctico que aparezca en cualquier otro lugar. Cuando se ejecuta la sentencia return, la función que la contiene devuelve el valor de la *expresión* a su invocador. Por ejemplo:

```
function square(x) { return x*x; } // Una función que tiene una declaración de  
// retorno  
square(2) // => 4
```

Sin declaración de retorno, una invocación a una función simplemente ejecuta cada una de las declaraciones en el cuerpo de la función por turnos hasta que llega al final de la función y luego regresa a su llamador. En este caso, la expresión de invocación se evalúa como indefinida. La sentencia return aparece a menudo como la última sentencia de una función, pero no tiene por qué ser la última: una función vuelve a su invocador cuando se ejecuta una sentencia return, incluso si quedan otras sentencias en el cuerpo de la función.

La sentencia return también se puede utilizar sin una *expresión* para que la función devuelva algo indefinido a quien la llama. Por ejemplo:

```
function displayObject(o) {  
    // Devuelve inmediatamente si el argumento es nulo o indefinido.  
    if (!o) return; // El resto de la función va aquí...  
}
```

Debido a la inserción automática de punto y coma de JavaScript ([§2.6](#)), no puede incluir un salto de línea entre la palabra clave return y la expresión que la sigue.

## 5.5.5 rendimiento

La sentencia `yield` es muy parecida a la sentencia `return`, pero sólo se utiliza en las funciones generadoras de ES6 (véase §12.3) para producir el siguiente valor en la secuencia de valores generada sin devolverlo realmente:

```
// Una función generadora que produce un rango de enteros function* range(from, to) { for(let i = from; i <= to; i++) { yield i; } }
```

Para entender `yield`, debe entender los iteradores y los generadores, que no se cubrirán hasta [el capítulo 12](#). Sin embargo, `yield` se incluye aquí para completar la información. (Sin embargo, técnicamente, `yield` es un operador más que una sentencia, como se explica en §12.4.2.)

## 5.5.6 lanzamiento

Una *excepción* es una señal que indica que se ha producido algún tipo de condición excepcional o error. *Lanzar una excepción* es *señalar* un error o una condición excepcional. *Atrapar una excepción* es manejarla, es decir, tomar cualquier acción necesaria o apropiada para recuperarse de la excepción. En JavaScript, las excepciones se lanzan siempre que se produce un error en tiempo de ejecución y siempre que el programa lanza una explícitamente utilizando la sentencia `throw`. Las excepciones se capturan con la sentencia `try/catch/finally`, que se describe en la siguiente sección.

La sentencia `throw` tiene la siguiente sintaxis:

*expresión de lanzamiento;*

puede evaluarse a un valor de cualquier tipo. Puede lanzar un número que represente un código de error o una cadena que contenga un mensaje de error legible para los humanos. La clase Error y sus subclases se utilizan cuando el propio intérprete de JavaScript lanza un error, y usted también puede utilizarlas. Un objeto Error tiene una propiedad name que especifica el tipo de error y una propiedad message que contiene la cadena pasada a la función constructora. A continuación se muestra una función de ejemplo que lanza un objeto Error cuando se invoca con un argumento no válido:

```
function factorial(x) { // Si el argumento de entrada no es válido, ¡lanza una excepción! if (x < 0) lanza un nuevo Error("x no debe ser negativo"); // En caso contrario, calcula un valor y devuelve normalmente let f; for(f = 1; x > 1; f *= x, x--) /* empty */; return f; }
factorial(4) //=> 24
```

Cuando se lanza una excepción, el intérprete de JavaScript detiene inmediatamente la ejecución normal del programa y salta al manejador de excepciones más cercano. Los manejadores de excepciones se escriben utilizando la cláusula catch de la sentencia try/catch/finally, que se describe en la siguiente sección. Si el bloque de código en el que se lanzó la excepción no tiene una cláusula catch asociada, el intérprete comprueba el siguiente bloque de código para ver si tiene un manejador de excepciones asociado. Esto continúa hasta que se encuentra un manejador. Si se lanza una excepción en una función que no contiene una sentencia try/catch/finally para manejarla, la excepción se propaga hasta el código que invocó la función. De este modo, las excepciones se propagan a través de la estructura léxica de los métodos de JavaScript y de la pila de llamadas. Si no se encuentra

ningún manejador de excepciones, la excepción se trata como un error y se informa al usuario.

### 5.5.7 try/catch/finally

La sentencia try/catch/finally es el mecanismo de manejo de excepciones de JavaScript. La cláusula try de esta sentencia simplemente define el bloque de código cuyas excepciones deben ser manejadas. El bloque try va seguido de una cláusula catch, que es un bloque de sentencias que se invoca cuando se produce una excepción en cualquier parte del bloque try. La cláusula catch va seguida de un bloque finally que contiene código de limpieza que se garantiza que se ejecutará, independientemente de lo que ocurra en el bloque try. Los bloques catch y finally son opcionales, pero un bloque try debe ir acompañado de al menos uno de estos bloques. Los bloques try, catch y finally comienzan y terminan con llaves. Estas llaves son una parte obligatoria de la sintaxis y no pueden omitirse, incluso si una cláusula contiene sólo una declaración.

El siguiente código ilustra la sintaxis y el propósito de la sentencia try/catch/finally:

```
intentar {  
    // Normalmente, este código se ejecuta desde la parte superior del bloque hasta la  
    // parte inferior  
    // sin problemas. Pero a veces puede lanzar una excepción,  
    // ya sea directamente, con una sentencia throw, o indirectamente,  
    // llamando // a un método que lance una excepción. } catch(e) {  
    // Las sentencias de este bloque se ejecutan si, y sólo si, el try  
    // el bloque lanza una excepción. Estas sentencias pueden utilizar la variable local // e  
    // para referirse al objeto Error u otro valor que se haya lanzado.  
    // Este bloque puede manejar la excepción de alguna manera, puede ignorar  
    // la // excepción sin hacer nada, o puede volver a lanzar la excepción con throw.  
  
} finalmente  
{  
    // Este bloque contiene sentencias que se ejecutan siempre, independientemente de  
    // lo que ocurre en el bloque try. Se ejecutan tanto si el bloque try  
    // el bloque termina:  
    // 1) normalmente, después de llegar al fondo del bloque  
    // 2) debido a una sentencia break, continue o return  
    // 3) con una excepción que es manejada por una cláusula catch anterior  
    // 4) con una excepción no capturada que se sigue propagando  
}
```

Tenga en cuenta que la palabra clave `catch` suele ir seguida de un identificador entre paréntesis. Este identificador es como un parámetro de la función. Cuando se atrapa una excepción, el valor asociado a la excepción (un objeto `Error`, por ejemplo) se asigna a este parámetro. El identificador asociado a una cláusula `catch` tiene alcance de bloque: sólo se define dentro del bloque `catch`.

Este es un ejemplo realista de la sentencia `try/catch`. Utiliza el método `factorial()` definido en la sección anterior y los métodos

JavaScript del lado del cliente `prompt()` y `alert()` para la entrada y la salida:

```
try { // Pide al usuario que introduzca un número
  let n = Number(prompt("Por favor, introduzca un número entero positivo",
  ""));
  // Calcula el factorial del número, asumiendo que la entrada es válida let f =
  factorial(n); // Muestra el resultado alert(n + "!" + f);
}

catch(ex) { // Si la entrada del usuario no fue válida, terminamos aquí alert(ex); //
Dile al usuario cuál es el error }
```

Este ejemplo es una sentencia `try/catch` sin cláusula `finally`.

Aunque `finally` no se utiliza tan a menudo como `catch`, puede ser útil. Sin embargo, su comportamiento requiere una explicación adicional. Se garantiza que la cláusula `finally` se ejecute si se ejecuta cualquier parte del bloque `try`, independientemente de cómo se complete el código del bloque `try`. Generalmente se utiliza para limpiar después del código en la cláusula `try`.

En el caso normal, el intérprete de JavaScript llega al final del bloque `try` y pasa al bloque `finally`, que realiza cualquier limpieza necesaria. Si el intérprete abandonó el bloque `try` debido a una sentencia `return`, `continue` o `break`, el bloque `finally` se ejecuta antes de que el intérprete salte a su nuevo destino.

Si se produce una excepción en el bloque `try` y hay un bloque `catch` asociado para manejar la excepción, el intérprete ejecuta primero el bloque `catch` y luego el bloque `finally`. Si no hay un bloque `catch`

local para manejar la excepción, el intérprete ejecuta primero el bloque finally y luego salta a la cláusula catch más cercana.

Si un bloque finally provoca un salto con una sentencia return, continue, break o throw, o llamando a un método que lanza una excepción, el intérprete abandona cualquier salto que estuviera pendiente y realiza el nuevo salto. Por ejemplo, si una cláusula finally lanza una excepción, esa excepción reemplaza cualquier excepción que estuviera en proceso de ser lanzada. Si una cláusula finally emite una sentencia return, el método retorna normalmente, incluso si se ha lanzado una excepción y aún no se ha manejado.

Los bloques try y finally pueden usarse juntos sin una cláusula catch. En este caso, el bloque finally es simplemente un código de limpieza que está garantizado para ser ejecutado, independientemente de lo que ocurra en el bloque try.

Recordemos que no podemos simular completamente un bucle for con un bucle while porque la sentencia continue se comporta de forma diferente para los dos bucles. Si añadimos una sentencia try/finally, podemos escribir un bucle while que funcione como un bucle for y que maneje correctamente las sentencias continue:

```
// Simular for(initialize ; test ; increment ) body; initialize ; while( test )
{ try { body ; } finally { increment ; } }
```

Tenga en cuenta, sin embargo, que un *cuerpo* que contiene una sentencia break se comporta de forma ligeramente diferente (provocando un incremento extra antes de salir) en el bucle while que en el bucle for, por lo que incluso con la cláusula finally, no es posible simular completamente el bucle for con while.

## CLÁUSULAS DE CAPTURA

Ocasionalmente es posible que te encuentres utilizando la cláusula únicamente para detectar y detener la propagación de una excepción, aunque no le importe el tipo o el valor de la excepción. En ES2019 y posteriores, puede omitir los paréntesis y el identificador y utilizar el atrapa palabra clave desnuda en caso. He aquí un ejemplo:

```
// Como JSON.parse(), pero devolviendo undefined en lugar de lanzar un error
función parseJSON(s) {
    pru{
        devolveJSON.analiz(s);
    } atrapa{
        // Algo salió mal pero no nos importa lo que fue
        devolvéndefinido;
    }
}
```

## 5.6 Declaraciones diversas

En esta sección se describen las tres declaraciones restantes de JavaScript -con, debugger, y "use strict".

### 5.6.1 con

La sentencia with ejecuta un bloque de código como si las propiedades de un objeto especificado fueran variables en el ámbito de ese código. Tiene la siguiente sintaxis:

con (*objeto*) *declaración*

Esta sentencia crea un ámbito temporal con las propiedades del *objeto* como variables y luego ejecuta *la sentencia* dentro de ese ámbito.

La sentencia with está prohibida en modo estricto (véase §5.6.3) y debe considerarse obsoleta en modo no estricto: evite su uso

siempre que sea posible. El código JavaScript que utiliza `with` es difícil de optimizar y es probable que se ejecute significativamente más lento que el código equivalente escrito sin la sentencia `with`.

El uso habitual de la sentencia `with` es facilitar el trabajo con jerarquías de objetos profundamente anidadas. En el lado del cliente de JavaScript, por ejemplo, puede tener que escribir expresiones como ésta para acceder a los elementos de un formulario HTML:

```
documento.formularios[0].dirección.valor
```

Si necesita escribir expresiones como ésta varias veces, puede utilizar la sentencia `with` para tratar las propiedades del objeto formulario como variables:

```
with(documento.forms[0]) { // Accede aquí  
directamente a los elementos del formulario. Por  
ejemplo:  
    name.value = ""; address.value = ""; email.value  
    = "";  
}
```

Esto reduce la cantidad de escritura que tiene que hacer: ya no necesita anteponer el nombre de cada propiedad del formulario con `document.forms[0]`. Es igual de sencillo, por supuesto, evitar la sentencia `with` y escribir el código anterior así:

```
let f = document.forms[0];  
f.nombre.valor = "";  
f.dirección.valor = "";  
f.correo electrónico.valor = "";
```

Tenga en cuenta que si utiliza `const` o `let` o `var` para declarar una variable o constante dentro del cuerpo de una sentencia `with`, crea

una variable ordinaria y no define una nueva propiedad dentro del objeto especificado.

## 5.6.2 depurador

La sentencia debugger normalmente no hace nada. Sin embargo, si un programa de depuración está disponible y se está ejecutando, entonces una implementación puede (pero no está obligada a) realizar algún tipo de acción de depuración. En la práctica, esta sentencia actúa como un punto de interrupción: la ejecución del código JavaScript se detiene, y se puede utilizar el depurador para imprimir los valores de las variables, examinar la pila de llamadas, etc. Supongamos, por ejemplo, que estás recibiendo una excepción en tu función f() porque está siendo llamada con un argumento indefinido, y no puedes averiguar de dónde viene esta llamada. Para ayudarte en la depuración de este problema, podrías alterar f() para que comience así:

```
function f(o) { if (o === undefined) debugger; // Línea temporal para fines de depuración ... // El resto de la función va aquí. }
```

Ahora, cuando se llame a f() sin argumento, la ejecución se detendrá, y podrá utilizar el depurador para inspeccionar la pila de llamadas y averiguar de dónde procede esta llamada incorrecta.

Ten en cuenta que no es suficiente con tener un depurador disponible: la sentencia debugger no iniciará el depurador por ti. Sin embargo, si estás utilizando un navegador web y tienes la consola de herramientas de desarrollo abierta, esta sentencia provocará un punto de interrupción.

## 5.6.3 "uso estricto"

"use strict" es una *directiva* introducida en ES5. Las directivas no son sentencias (pero se acercan lo suficiente como para que "use strict" se documente aquí). Hay dos diferencias importantes entre la directiva "use strict" y las sentencias normales:

- No incluye ninguna palabra clave del lenguaje: la directiva es sólo una declaración de expresión que consiste en un literal de cadena especial (entre comillas simples o dobles).
- Sólo puede aparecer al principio de un script o al principio del cuerpo de una función, antes de que aparezca cualquier declaración real.

El propósito de una directiva "use strict" es indicar que el código que sigue (en el script o la función) es *código estricto*. El código de nivel superior (no función) de un script es código estricto si el script tiene una directiva "use strict". El cuerpo de una función es código estricto si se define dentro de código estricto o si tiene una directiva "use strict". El código que se pasa al método eval() es código estricto si eval() se llama desde código estricto o si la cadena de código incluye una directiva "use strict". Además del código declarado explícitamente como estricto, cualquier código en el cuerpo de una clase ([Capítulo 9](#)) o en un módulo ES6 ([§10.3](#)) es automáticamente código estricto. Esto significa que si todo tu código JavaScript está escrito como módulos, entonces todo es automáticamente estricto, y nunca necesitarás usar una directiva explícita "use strict".

El código estricto se ejecuta en *modo estricto*. El modo estricto es un subconjunto restringido del lenguaje que corrige importantes deficiencias del mismo y proporciona una mayor comprobación de

errores y una mayor seguridad. Dado que el modo estricto no es el predeterminado, el código JavaScript antiguo que aún utiliza las características heredadas deficientes del lenguaje seguirá ejecutándose correctamente. Las diferencias entre el modo estricto y el modo no estricto son las siguientes (las tres primeras son especialmente importantes):

- La sentencia `with` no está permitida en modo estricto.
- En modo estricto, todas las variables deben ser declaradas: a
  - Se lanza un `ReferenceError` si se asigna un valor a un identificador que no es una variable declarada, función, parámetro de función, parámetro de cláusula `catch` o propiedad del objeto global. (En modo no estricto, esto declara implícitamente una variable global añadiendo una nueva propiedad al objeto global).
- En modo estricto, las funciones invocadas como funciones (en lugar de como métodos) tienen un valor `this` de indefinido. (En modo no estricto, a las funciones invocadas como funciones siempre se les pasa el objeto global como su valor `this`). Además, en modo estricto, cuando una función es invocada con `call()` o `apply()` (§8.7.4), el valor `this` es exactamente el valor pasado como primer argumento a `call()` o `apply()`. (En modo no estricto, los valores nulos e indefinidos se sustituyen por el objeto global y los valores no objetuales se convierten en objetos).
- En modo estricto, las asignaciones a propiedades no escribibles y los intentos de crear nuevas propiedades en

objetos no extensibles arrojan un `TypeError`. (En modo no estricto, estos intentos fallan silenciosamente).

- En el modo estricto, el código pasado a `eval()` no puede declarar variables o definir funciones en el ámbito de la persona que llama, como puede hacerlo en el modo no estricto. En su lugar, las definiciones de variables y funciones viven en un nuevo ámbito creado para `eval()`. Este ámbito se descarta cuando `eval()` regresa.
- En modo estricto, el objeto `Arguments` ([§8.3.3](#)) de una función contiene una copia estática de los valores pasados a la función. En modo no estricto, el objeto `Arguments` tiene un comportamiento "mágico" en el que los elementos de la matriz y los parámetros de la función con nombre se refieren al mismo valor.
- En modo estricto, se lanza un `SyntaxError` si el operador de borrado va seguido de un identificador no cualificado como una variable, función o parámetro de función. (En modo no estricto, una expresión de borrado de este tipo no hace nada y se evalúa a `false`).
- En modo estricto, un intento de borrar una propiedad no configurable arroja un `TypeError`. (En modo no estricto, el intento falla y la expresión de borrado se evalúa como falsa).
- En modo estricto, es un error de sintaxis que un literal de objeto defina dos o más propiedades con el mismo nombre. (En modo no estricto, no se produce ningún error).
- En modo estricto, es un error de sintaxis que una declaración de función tenga dos o más parámetros con el

mismo nombre. (En modo no estricto, no se produce ningún error).

- En modo estricto, no se permiten los literales octales (que comienzan con un 0 y no van seguidos de una x). (En modo no estricto, algunas implementaciones permiten los literales octales).
- En modo estricto, los identificadores eval y arguments son tratados como palabras clave, y no se permite cambiar su valor. No puede asignar un valor a estos identificadores, declararlos como variables, utilizarlos como nombres de funciones, utilizarlos como nombres de parámetros de funciones o utilizarlos como identificador de un bloque catch.
- En modo estricto, la capacidad de examinar la pila de llamadas está restringida. arguments.caller y argumentscallee lanzan un TypeError dentro de una función de modo estricto. Las funciones de modo estricto también tienen propiedades caller y arguments que lanzan TypeError cuando se leen. (Algunas implementaciones definen estas propiedades no estándar en funciones no estrictas).

## 5.7 Declaraciones

Las palabras clave const, let, var, function, class, import y export no son técnicamente sentencias, pero se parecen mucho a las sentencias, y este libro se refiere informalmente a ellas como sentencias, por lo que merecen una mención en este capítulo.

Estas palabras clave se describen mejor como *declaraciones* que como afirmaciones. Dijimos al principio de este capítulo que las

declaraciones "hacen que algo suceda". Las declaraciones sirven para definir nuevos valores y darles nombres que podemos usar para referirnos a esos valores. No hacen que ocurra mucho por sí mismas, pero al proporcionar nombres para los valores, en un sentido importante, definen el significado de las otras declaraciones en su programa.

Cuando un programa se ejecuta, son las expresiones del programa las que se evalúan y las declaraciones del programa las que se ejecutan. Las declaraciones de un programa no se "ejecutan" de la misma manera: en su lugar, definen la estructura del propio programa. En términos generales, puedes pensar en las declaraciones como las partes del programa que se procesan antes de que el código comience a ejecutarse.

Las declaraciones de JavaScript se utilizan para definir constantes, variables, funciones y clases y para importar y exportar valores entre módulos. Las siguientes subsecciones dan ejemplos de todas estas declaraciones. Todas ellas se tratan con mucho más detalle en otras partes de este libro.

### 5.7.1 const, let y var

Las declaraciones const, let y var se tratan en [§3.10](#). En ES6 y posteriores, const declara constantes, y let declara variables. Antes de ES6, la palabra clave var era la única forma de declarar variables y no había forma de declarar constantes. Las variables declaradas con var se asignan a la función que las contiene y no al bloque que las contiene. Esto puede ser una fuente de errores, y en el JavaScript moderno no hay realmente ninguna razón para usar var en lugar de let.

```
const TAU = 2*Math.PI; let radius = 3;  
var circunferencia = TAU * radio;
```

## 5.7.2 función

La declaración de función se utiliza para definir funciones, que se tratan en detalle en [el capítulo 8](#). (También vimos la función en [§4.3](#), donde se utilizó como parte de una expresión de función en lugar de una declaración de función). Una declaración de función tiene el siguiente aspecto:

```
function area(radius) { return Math.PI * radio *  
    radio; }
```

Una declaración de función crea un objeto de función y lo asigna al nombre especificado (área en este ejemplo). En cualquier parte de nuestro programa, podemos referirnos a la función -y ejecutar el código dentro de ella- utilizando este nombre. Las declaraciones de funciones en cualquier bloque de código JavaScript se procesan antes de que se ejecute el código, y los nombres de las funciones se vinculan a los objetos de función en todo el bloque. Decimos que las declaraciones de funciones son "elevadas" porque es como si se hubieran movido a la parte superior del ámbito en el que están definidas. El resultado es que el código que invoca una función puede existir en su programa antes del código que declara la función.

En [§12.3](#) se describe un tipo especial de función conocida como *generador*. Las declaraciones de generadores utilizan la palabra clave `function` pero van seguidas de un asterisco. En [§13.3](#) se describen las funciones asíncronas, que también se declaran utilizando la palabra clave `function` pero con el prefijo `async`.

### 5.7.3 clase

En ES6 y posteriores, la declaración de clase crea una nueva clase y le da un nombre que podemos utilizar para referirnos a ella. Las clases se describen en detalle en el [Capítulo 9](#). Una declaración de clase simple puede ser como esta:

```
class Círculo { constructor(radio) { this.r = radio; } área() { return  
    Math.PI * this.r * this.r; } circunferencia() { devuelve 2 * Math.PI *  
    this.r; } }
```

A diferencia de las funciones, las declaraciones de clases no son hoisted, y no se puede utilizar una clase declarada de esta manera en el código que aparece antes de la declaración.

### 5.7.4 importación y exportación

Las declaraciones "import" y "export" se utilizan conjuntamente para que los valores definidos en un módulo de código JavaScript estén disponibles en otro módulo. Un módulo es un archivo de código JavaScript con su propio espacio de nombres global, completamente independiente de todos los demás módulos. La única manera de que un valor (como una función o una clase) definido en un módulo pueda ser utilizado en otro módulo es si el módulo que lo define lo exporta con `export` y el módulo que lo utiliza lo importa con `import`. Los módulos son el tema del [capítulo 10](#), y la importación y la exportación se tratan en detalle en [§10.3](#).

Las directivas `import` se utilizan para importar uno o más valores de otro archivo de código JavaScript y darles nombres dentro del módulo actual.

Las directivas de importación tienen diferentes formas. He aquí algunos ejemplos:

```
import Circle from './geometry/circle.js'; import { PI, TAU } from
'./geometry/constants.js'; import { magnitude as hypotenuse } from
'./vectors/utils.js';
```

Los valores dentro de un módulo de JavaScript son privados y no pueden ser importados en otros módulos a menos que hayan sido exportados explícitamente. La directiva `export` hace esto: declara que uno o más valores definidos en el módulo actual son exportados y por lo tanto están disponibles para ser importados por otros módulos. La directiva `export` tiene más variantes que la directiva `import`. Esta es una de ellas:

```
// geometry/constants.js
const PI = Math.PI; const TAU
= 2 * PI; export { PI, TAU };
```

La palabra clave `export` se utiliza a veces como modificador de otras declaraciones, dando lugar a una especie de declaración compuesta que define una constante, variable, función o clase y la exporta al mismo tiempo. Y cuando un módulo exporta sólo un valor, se suele hacer con la forma especial `export default`:

```
export const TAU = 2 * Math.PI; export function magnitude(x,y) { return Math.
sqrt(x*x + y*y); } export default class Circle { /* class definition omitted here */ }
```

## 5.8 Resumen de las declaraciones de JavaScript

Este capítulo ha introducido cada una de las sentencias del lenguaje JavaScript, que se resumen en la [Tabla 5-1](#).

*Tabla 5-1. Sintaxis de las sentencias de JavaScript*

| Declaración    | Propósito   |
|----------------|---|
| romper         | Salir del bucle o conmutador más interno o de la sentencia adjunta con nombre |
| caso           | Etiquetar una sentencia dentro de un switch                                   |
| clase          | Declarar una clase  |
| const          | Declarar e inicializar una o más constantes                                   |
| continuar      | Comienza la siguiente iteración del bucle más interno o del bucle con nombre  |
| depurador      | Punto de interrupción del depurador   |
| por defecto    | Etiquetar la sentencia por defecto dentro de un interruptor                   |
| hacer/mientras | Una alternativa al bucle while  |
| exportDeclara  | valores que pueden ser importados en otros módulos                            |
| para           | Un bucle fácil de usar  |
| para/espera    | Iterar de forma asíncrona los valores de un iterador asíncrono                |
| para/en        | Enumerar los nombres de las propiedades de un objeto                          |
| para/de        | Enumerar los valores de un objeto iterable como un array                      |
| función        | Declarar una función  |
| si/no          | Ejecutar una sentencia u otra en función de una condición                     |

|                       |  |
|-----------------------|--|
| importar              | Declarar nombres para valores definidos en otros módulos                             |
| etiqueta              | Dar un nombre a la sentencia para usarla con break y continue                        |
| dejar                 | Declarar e inicializar una o más variables de ámbito de bloque (nueva sintaxis)      |
| devolver              | Devolver un valor de una función   |
| cambiar               | Bifurcación multidireccional a caso o por defecto: etiquetas                         |
| tirar                 | Lanzar una excepción   |
| try/catch/final<br>ly | Gestionar las excepciones y la limpieza del código                                   |
| "uso estricto"        | Aplicar restricciones de modo estricto al script o a la función                      |
| var                   | Declarar e inicializar una o más variables (sintaxis antigua)                        |
| mientras que          | Una construcción de bucle básica   |
| con                   | Ampliar la cadena de alcance (obsoleto y prohibido en modo estricto)                 |
| rendimiento           | Proporcionar un valor para ser iterado; sólo se utiliza en las funciones generadoras |

El hecho de que las expresiones case se evalúen en tiempo de ejecución hace que el JavaScript

<sup>1</sup> muy diferente (y menos eficiente) que la sentencia switch de C, C++ y Java. En esos lenguajes, las expresiones de caso deben ser constantes en tiempo de compilación del mismo tipo, y las sentencias switch a menudo pueden compilar hasta *tablas de salto* altamente eficientes.

Cuando consideremos la sentencia continue en §5.5.3, veremos que este bucle while no es un equivalente exacto del bucle for.



# Capítulo 6. Objetos

---

Los objetos son el tipo de datos más fundamental de JavaScript, y ya los has visto muchas veces en los capítulos que preceden a éste. Debido a que los objetos son tan importantes para el lenguaje JavaScript, es importante que entiendas cómo funcionan en detalle, y este capítulo proporciona ese detalle. Comienza con un resumen formal de los objetos, y luego se sumerge en secciones prácticas sobre la creación de objetos y la consulta, configuración, eliminación, prueba y enumeración de las propiedades de los objetos. Estas secciones centradas en las propiedades van seguidas de otras que explican cómo extender, serializar y definir métodos importantes en los objetos. Finalmente, el capítulo concluye con una larga sección sobre la nueva sintaxis literal de objetos en ES6 y en las versiones más recientes del lenguaje.

## 6.1 Introducción a los objetos

Un objeto es un valor compuesto: agrega múltiples valores (valores primitivos u otros objetos) y permite almacenar y recuperar esos valores por su nombre. Un objeto es una colección desordenada de *propiedades*, cada una de las cuales tiene un nombre y un valor. Los nombres de las propiedades suelen ser cadenas (aunque, como veremos en §6.10.3, los nombres de las propiedades también pueden ser símbolos), por lo que podemos decir que los objetos asignan cadenas a valores. Este mapeo de cadenas a

valores recibe varios nombres -probablemente ya estés familiarizado con la estructura de datos fundamental bajo el nombre de "hash", "hashtable", "diccionario" o "array asociativo". Sin embargo, un objeto es más que un simple mapa de cadena a valor. Además de mantener su propio conjunto de propiedades, un objeto de JavaScript también hereda las propiedades de otro objeto, conocido como su "prototipo". Los métodos de un objeto suelen ser propiedades heredadas, y esta "herencia prototípica" es una característica clave de JavaScript.

Los objetos de JavaScript son dinámicos -las propiedades suelen poder añadirse y eliminarse-, pero pueden utilizarse para simular los objetos estáticos y los "structs" de los lenguajes de tipado estático. También pueden utilizarse (ignorando la parte del valor del mapeo cadena-valor) para representar conjuntos de cadenas.

Cualquier valor en JavaScript que no sea una cadena, un número, un símbolo, o verdadero, falso, nulo o indefinido es un objeto. Y aunque las cadenas, los números y los booleanos no son objetos, pueden comportarse como objetos inmutables.

Recordemos del §3.8 que los objetos son *mutables* y se manipulan por referencia en lugar de por valor. Si la variable x se refiere a un objeto y se ejecuta el código let y = x; la variable y contiene una referencia al mismo objeto, no una copia de ese objeto. Cualquier modificación hecha al objeto a través de la variable y es también visible a través de la variable x.

Las cosas más comunes que se hacen con los objetos son crearlos y establecer, consultar, eliminar, probar y enumerar sus

propiedades. Estas operaciones fundamentales se describen en las primeras secciones de este capítulo. Las secciones siguientes cubren temas más avanzados.

Una *propiedad* tiene un nombre y un valor. El nombre de una propiedad puede ser cualquier cadena, incluyendo la cadena vacía (o cualquier símbolo), pero ningún objeto puede tener dos propiedades con el mismo nombre. El valor puede ser cualquier valor de JavaScript, o puede ser una función getter o setter (o ambas).

Aprenderemos sobre las funciones getter y setter en [§6.10.6](#).

A veces es importante saber distinguir entre las propiedades definidas directamente en un objeto y las que se heredan de un objeto prototipo. JavaScript utiliza el término *propiedad propia* para referirse a las propiedades no heredadas.

Además de su nombre y valor, cada propiedad tiene tres *atributos de propiedad*:

- El atributo *writable* especifica si el valor de la propiedad puede ser establecido.
- El atributo *enumerable* especifica si el nombre de la propiedad es devuelto por un bucle for/in.
- El atributo *configurable* especifica si la propiedad puede ser eliminada y si sus atributos pueden ser alterados.

Muchos de los objetos incorporados de JavaScript tienen propiedades que son de sólo lectura, no enumerables o no configurables. Sin embargo, por defecto, todas las propiedades de los objetos que creas son escribibles, enumerables y configurables.

En §14.1 se explican técnicas para especificar valores de atributos de propiedades no predeterminados para tus objetos.

## 6.2 Crear objetos

Los objetos pueden crearse con literales de objeto, con la palabra clave new y con la función Object.create(). Las subsecciones siguientes describen cada técnica.

### 6.2.1 Literales de objeto

La forma más sencilla de crear un objeto es incluir un literal de objeto en el código JavaScript. En su forma más simple, un literal de *objeto* es una lista separada por comas de pares nombre:valor separados por dos puntos, encerrados entre llaves. Un nombre de propiedad es un identificador de JavaScript o un literal de cadena (se permite la cadena vacía). Un valor de propiedad es cualquier expresión de JavaScript; el valor de la expresión (puede ser un valor primitivo o un valor de objeto) se convierte en el valor de la propiedad. He aquí algunos ejemplos:

```
let empty = {}; // Un objeto sin propiedades
let point = { x: 0, y: 0 }; // Dos propiedades numéricas
let p2 = { x: punto.x, y: punto.y+1 }; // Valores más complejos
let book = {
    "título principal": "JavaScript", // Estos nombres de propiedades incluyen espacios,
    "subtítulo": "The Definitive Guide", // y guiones, por lo que hay que utilizar literales de cadena.
    para: "todos los públicos", // for está reservado, pero sin comillas.
    autor: { // El valor de esta propiedad es firstname: "David", // en sí mismo un objeto.
        apellido: "Flanagan" }
};
```

Una coma al final de la última propiedad en un literal de objeto es legal, y algunos estilos de programación fomentan el uso de estas comas al final para que sea menos probable que se produzca un error de sintaxis si se añade una nueva propiedad al final del literal de objeto en algún momento posterior.

Un literal de objeto es una expresión que crea e inicializa un objeto nuevo y distinto cada vez que se evalúa. El valor de cada propiedad se evalúa cada vez que se evalúa el literal. Esto significa que un solo literal de objeto puede crear muchos objetos nuevos si aparece dentro del cuerpo de un bucle o en una función a la que se llama repetidamente, y que los valores de las propiedades de estos objetos pueden ser diferentes entre sí.

Los literales de objeto mostrados aquí utilizan una sintaxis simple que ha sido legal desde las primeras versiones de JavaScript. Las versiones recientes del lenguaje han introducido una serie de

nuevas características de los literales de objeto, que se tratan en [§6.10](#).

### 6.2.2 Creación de objetos con nuevos

El operador new crea e inicializa un nuevo objeto. La palabra clave new debe ir seguida de una invocación a una función. Una función utilizada de este modo se denomina *constructor* y sirve para inicializar un objeto recién creado. JavaScript incluye constructores para sus tipos incorporados. Por ejemplo:

```
let o = new Object(); // Crear un objeto vacío: igual que {}. let a = new Array(); //  
Crear un array vacío: igual que []. let d = new Date(); // Crear un objeto Date que  
represente la hora actual let r = new Map(); // Crear un objeto Map para la  
asignación de claves/valores
```

Además de estos constructores incorporados, es común definir sus propias funciones constructoras para inicializar los objetos recién creados. Esto se explica en el [Capítulo 9](#).

### 6.2.3 Prototipos

Antes de que podamos cubrir la tercera técnica de creación de objetos, debemos detenernos un momento para explicar los prototipos. Casi todos los objetos de JavaScript tienen un segundo objeto de JavaScript asociado. Este segundo objeto se conoce como *prototipo*, y el primer objeto hereda propiedades del prototipo.

Todos los objetos creados por los literales de objeto tienen el mismo objeto prototipo, y podemos referirnos a este objeto prototipo en el código JavaScript como

Objeto.prototype. Los objetos creados mediante la palabra clave new y la invocación de un constructor utilizan el valor de la propiedad prototype de la función constructora como su prototipo. Así, el objeto creado por new Object() hereda de Object.prototype, al igual que el objeto creado por {}. Del mismo modo, el objeto creado por new Array() utiliza Array.prototype como su prototipo, y el objeto creado por new Date() utiliza Date.prototype como su prototipo.

Esto puede ser confuso cuando se aprende JavaScript por primera vez. Recuerda: casi todos los objetos tienen un *prototipo*, pero sólo un número relativamente pequeño de objetos tienen una propiedad de prototipo. Son estos objetos con propiedades de prototipo los que definen los *prototipos* de todos los demás objetos.

Object.prototype es uno de los raros objetos que no tiene prototipo: no hereda ninguna propiedad. Otros objetos prototipo son objetos normales que sí tienen un prototipo. La mayoría de los constructores incorporados (y la mayoría de los constructores definidos por el usuario) tienen un prototipo que hereda de Object.prototype. Por ejemplo, Date.prototype hereda propiedades de Object.prototype, por lo que un objeto Date creado por new Date() hereda propiedades tanto de Date.prototype como de Object.prototype. Esta serie enlazada de objetos prototipo se conoce como *cadena de prototipos*.

En [§6.3.2](#) se explica cómo funciona la herencia de propiedades. El [capítulo 9](#) explica la conexión entre prototipos y constructores con más detalle: muestra cómo definir nuevas "clases" de objetos escribiendo una función constructora y estableciendo su

propiedad prototipo al objeto prototipo que utilizarán las "instancias" creadas con ese constructor. Y aprenderemos cómo consultar (e incluso cambiar) el prototipo de un objeto en [§14.3](#).

## 6.2.4 Object.create()

Object.create() crea un nuevo objeto, utilizando su primer argumento como el prototipo de ese objeto:

```
let o1 = Object.create({x: 1, y: 2}); // o1 hereda las propiedades x e y. o1.x +  
o1.y //=> 3
```

Puedes pasar null para crear un nuevo objeto que no tenga un prototipo, pero si haces esto, el objeto recién creado no heredará nada, ni siquiera métodos básicos como `toString()` (lo que significa que tampoco funcionará con el operador `+`):

```
let o2 = Object.create(null); // o2 no hereda ninguna propiedad o método.
```

Si quieres crear un objeto vacío ordinario (como el objeto devuelto por `{}` o `new Object()`), pasa `Object.prototype`:

```
let o3 = Object.create(Object.prototype); // o3 es como {} o new Object().
```

La capacidad de crear un nuevo objeto con un prototipo arbitrario es muy poderosa, y utilizaremos `Object.create()` en varios lugares a lo largo de este capítulo. (`Object.create()` también toma un segundo argumento opcional que describe las propiedades del nuevo objeto. Este segundo argumento es una característica avanzada cubierta en [§14.1](#).)

Uno de los usos de `Object.create()` es cuando se quiere evitar la modificación involuntaria (pero no maliciosa) de un objeto por una función de la biblioteca sobre la que no se tiene control. En lugar

de pasar el objeto directamente a la función, puede pasar un objeto que herede de él. Si la función lee las propiedades de ese objeto, verá los valores heredados. Sin embargo, si establece propiedades, esas escrituras no afectarán al objeto original.

```
let o = { x: "no cambies este valor" }; library.function(Object.create(o)); // Guarda contra modificaciones accidentales
```

Para entender por qué esto funciona, es necesario saber cómo se consultan y establecen las propiedades en JavaScript. Estos son los temas de la siguiente sección.

## 6.3 Consulta y configuración de propiedades

Para obtener el valor de una propiedad, utilice el punto (.) o el corchete

([]) descritos en §4.4. El lado izquierdo debe ser una expresión cuyo valor sea un objeto. Si se utiliza el operador punto, el lado derecho debe ser un identificador simple que nombre la propiedad. Si se utilizan corchetes, el valor dentro de los corchetes debe ser una expresión que se evalúe a una cadena que contenga el nombre de la propiedad deseada:

```
let author = book.author; // Obtener la propiedad "author" del libro.  
let name = author.surname; // Obtener la propiedad "surname" del autor.  
let title = book["main title"]; // Obtener la propiedad "main title" del libro.
```

Para crear o establecer una propiedad, utilice un punto o corchetes como lo haría para consultar la propiedad, pero póngalos en el lado izquierdo de una expresión de asignación:

```
libro.edición = 7; // Crear una propiedad "edición" del libro.  
book["main title"] = "ECMAScript"; // Cambiar la propiedad "main title".
```

Al utilizar la notación de corchetes, hemos dicho que la expresión dentro de los corchetes debe evaluarse a una cadena. Una afirmación más precisa es que la expresión debe evaluarse a una cadena o a un valor que pueda convertirse a una cadena o a un símbolo ([§6.10.3](#)). En el [capítulo 7](#), por ejemplo, veremos que es habitual utilizar números dentro de los corchetes.

### 6.3.1 Objetos como matrices asociativas

Como se ha explicado en la sección anterior, las dos expresiones JavaScript siguientes tienen el mismo valor:

```
objeto.propiedad objeto["propiedad"]
```

La primera sintaxis, que utiliza el punto y un identificador, es como la sintaxis utilizada para acceder a un campo estático de una estructura o un objeto en C o Java. La segunda sintaxis, que utiliza corchetes y una cadena, se parece al acceso a un array, pero a un array indexado por cadenas en lugar de por números. Este tipo de matriz se conoce como *matriz asociativa* (o hash o mapa o diccionario). Los objetos de JavaScript son matrices asociativas, y esta sección explica por qué es importante.

En C, C++, Java y otros lenguajes fuertemente tipados, un objeto sólo puede tener un número fijo de propiedades, y los nombres de estas propiedades deben definirse de antemano. Dado que JavaScript es un lenguaje poco tipado, esta regla no se aplica: un programa puede crear cualquier número de propiedades en cualquier objeto. Sin embargo, cuando se utiliza el operador `.` para acceder a una propiedad de un objeto, el nombre de la propiedad se expresa como un identificador. Los identificadores deben

escribirse literalmente en su programa JavaScript; no son un tipo de datos, por lo que no pueden ser manipulados por el programa.

Por otro lado, cuando se accede a una propiedad de un objeto con la notación de matriz [], el nombre de la propiedad se expresa como una cadena. Las cadenas son tipos de datos de JavaScript, por lo que pueden ser manipuladas y creadas mientras se ejecuta un programa. Así, por ejemplo, se puede escribir el siguiente código en JavaScript:

```
deje addr = "";  
  
for(let i = 0; i < 4; i++) { addr += cliente[`dress${i}`] + "\n";  
}
```

Este código lee y concatena las propiedades address0, address1, address2 y address3 del objeto cliente.

Este breve ejemplo demuestra la flexibilidad de utilizar la notación de matriz para acceder a las propiedades de un objeto con expresiones de cadena. Este código podría reescribirse utilizando la notación de puntos, pero hay casos en los que sólo sirve la notación de matrices. Suponga, por ejemplo, que está escribiendo un programa que utiliza recursos de red para calcular el valor actual de las inversiones bursátiles del usuario. El programa permite al usuario introducir el nombre de cada acción que posee, así como el número de acciones de cada una de ellas. Puede utilizar un objeto llamado cartera para mantener esta información. El objeto tiene una propiedad para cada acción. El nombre de la propiedad es el nombre de la acción, y el valor de la propiedad es el número de acciones de esa acción. Así, por ejemplo, si un

usuario tiene 50 acciones de IBM, la propiedad portfolio.ibm tiene el valor 50.

Parte de este programa podría ser una función para añadir una nueva acción a la cartera:

```
function addstock(portfolio, stockname, shares) {  
    portfolio[stockname] = shares; }
```

Dado que el usuario introduce los nombres de las acciones en tiempo de ejecución, no hay forma de conocer los nombres de las propiedades de antemano. Como no puede conocer los nombres de las propiedades cuando escribe el programa, no hay forma de que pueda utilizar el operador . para acceder a las propiedades del objeto cartera. Sin embargo, puede utilizar el operador [], porque utiliza un valor de cadena (que es dinámico y puede cambiar en tiempo de ejecución) en lugar de un identificador (que es estático y debe ser codificado en el programa) para nombrar la propiedad.

En el [capítulo 5](#), introdujimos el bucle for/in (y lo volveremos a ver en breve, en [§6.6](#)). El poder de esta sentencia de JavaScript queda claro cuando se considera su uso con matrices asociativas. Así es como se usaría para calcular el valor total de una cartera:

```
function computeValue(portfolio) { let total = 0.0; for(let stock in portfolio) { //  
    Para cada acción de la cartera:  
        let shares = portfolio[stock]; // obtener el número de acciones let price =  
        getPrice(stock); // buscar el precio de la acción total += shares * price; // añadir el  
        valor de la acción al valor total } return total; // devolver el valor total. }
```

Los objetos de JavaScript se utilizan comúnmente como matrices asociativas, como se muestra aquí, y es importante entender cómo

funciona. Sin embargo, en ES6 y posteriores, la clase Map descrita en [§11.1.2](#) es a menudo una mejor opción que utilizar un objeto simple.

### 6.3.2 Herencia

Los objetos de JavaScript tienen un conjunto de "propiedades propias", y también heredan un conjunto de propiedades de su objeto prototipo. Para entender esto, debemos considerar el acceso a las propiedades con más detalle. Los ejemplos de esta sección utilizan la función `Object.create()` para crear objetos con prototipos específicos. Sin embargo, veremos en el [capítulo 9](#) que cada vez que se crea una instancia de una clase con `new`, se está creando un objeto que hereda propiedades de un objeto prototipo.

Supongamos que se consulta la propiedad `x` en el objeto `o`. Si `o` no tiene una propiedad propia con ese nombre, se busca la propiedad `x` en el objeto prototipo de `o1`. Si el objeto prototipo no tiene una propiedad propia con ese nombre, pero tiene un prototipo propio, la consulta se realiza en el prototipo del prototipo. Esto continúa hasta que se encuentra la propiedad `x` o hasta que se busca un objeto con un prototipo nulo. Como puede ver, el atributo `prototipo` de un objeto crea una cadena o lista enlazada de la que se heredan propiedades:

```
let o = {};// o hereda los métodos del objeto de Object.prototype
o.x = 1;// y ahora tiene una propiedad propia x.
let p = Object.create(o);// p hereda las propiedades de o y Object.prototype
p.y = 2;// y tiene una propiedad propia y.
let q = Object.create(p);// q hereda propiedades de p, o, y...
q.z = 3;// ...Object.prototype y tiene una propiedad propia z. let f = q.toString(); //
toString se hereda de Object.prototype
q.x + q.y //=> 3; x e y se heredan de o y p
```

Ahora supongamos que se asigna a la propiedad x del objeto o. Si o ya tiene una propiedad propia (no heredada) llamada x, entonces la asignación simplemente cambia el valor de esta propiedad existente. En caso contrario, la asignación crea una nueva propiedad llamada x en el objeto o. Si o heredaba previamente la propiedad x, esa propiedad heredada queda ahora oculta por la propiedad propia recién creada con el mismo nombre.

La asignación de propiedades examina la cadena de prototipos sólo para determinar si la asignación está permitida. Si o hereda una propiedad de sólo lectura llamada x, por ejemplo, entonces la asignación no está permitida. (Los detalles sobre cuándo se puede establecer una propiedad se encuentran en §6.3.3.) Sin embargo, si la asignación está permitida, siempre crea o establece una propiedad en el objeto original y nunca modifica los objetos de la cadena de prototipos. El hecho de que la herencia se produzca al consultar las propiedades pero no al establecerlas es una característica clave de JavaScript porque nos permite anular selectivamente las propiedades heredadas:

```
let unitcircle = { r: 1 }; // Un objeto del que heredar let c = Object.create(unitcircle);
// c hereda la propiedad r
c. x = 1; c. y = 1; // c define dos propiedades propias
c. r = 2; // c anula su propiedad heredada unitcircle. r // => 1: el prototipo no se ve
afectado
```

Hay una excepción a la regla de que una asignación de propiedad falla o crea o establece una propiedad en el objeto original. Si o hereda la propiedad x, y esa propiedad es una propiedad accesoria con un método setter (véase §6.10.6), entonces se llama a ese método setter en lugar de crear una nueva propiedad x en o.

Tenga en cuenta, sin embargo, que el método setter se llama en el objeto o, no en el objeto prototipo que define la propiedad, por lo que si el método setter define alguna propiedad, lo hará en o, y volverá a dejar la cadena del prototipo sin modificar.

### 6.3.3 Errores de acceso a la propiedad

Las expresiones de acceso a propiedades no siempre devuelven o establecen un valor. Esta sección explica las cosas que pueden salir mal cuando se consulta o se establece una propiedad.

No es un error consultar una propiedad que no existe. Si la propiedad x no se encuentra como propiedad propia o heredada de o, la expresión de acceso a la propiedad o.x se evalúa como indefinida. Recordemos que nuestro objeto libro tiene una propiedad "subtítulo", pero no una propiedad "subtítulo":

```
libro. subtítulo // => undefined: la propiedad no existe
```

Sin embargo, es un error intentar consultar una propiedad de un objeto que no existe. Los valores null y undefined no tienen

propiedades, y es un error consultar las propiedades de estos valores. Continuando con el ejemplo anterior:

```
let len = book.subtitle.length; // !TypeError: undefined doesn't have length
```

Las expresiones de acceso a propiedades fallarán si el lado izquierdo de . es nulo o indefinido. Por lo tanto, al escribir una expresión como libro.autor.apellido, debe tener cuidado si no está seguro de que libro y libro.autor están realmente definidos. Aquí hay dos maneras de evitar este tipo de problemas:

```
// Una técnica verbose y explícita let surname =  
undefined; if (book) { if (book.author) { surname =  
book.author.surname; }  
}
```

```
// Una alternativa concisa e idiomática para obtener el apellido o null o undefined  
surname = book && book.author && book.author.surname;
```

Para entender por qué esta expresión idiomática funciona para evitar las excepciones de TypeError, puede que quiera revisar el comportamiento de cortocircuito del operador && en §4.10.1.

Como se describe en §4.4.1, ES2020 admite el acceso condicional a propiedades con ?, lo que nos permite reescribir la expresión de asignación anterior como

```
dejar apellido = libro?.autor?.apellido;
```

El intento de establecer una propiedad en un valor nulo o indefinido también provoca un TypeError. Los intentos de establecer propiedades sobre otros valores no siempre tienen éxito: algunas propiedades son de sólo lectura y no pueden ser

establecidas, y algunos objetos no permiten la adición de nuevas propiedades. En modo estricto ([§5.6.3](#)), se lanza un `TypeError` cada vez que falla un intento de establecer una propiedad. Fuera del modo estricto, estos fallos suelen ser silenciosos.

Las reglas que especifican cuándo una asignación de propiedad tiene éxito y cuándo falla son intuitivas pero difíciles de expresar de forma concisa. Un intento de establecer una propiedad `p` de un objeto `o` falla en estas circunstancias:

- `o` tiene una propiedad propia `p` que es de sólo lectura: no es posible establecer propiedades de sólo lectura.
- `o` tiene una propiedad heredada `p` que es de sólo lectura: no es posible ocultar una propiedad heredada de sólo lectura con una propiedad propia del mismo nombre.
- `o` no tiene una propiedad propia `p`; `o` no hereda una propiedad `p` con un método setter, y el atributo `extensible` de `o` ([véase §14.2](#)) es falso. Como `p` no existe ya en `o`, y si no hay un método setter al que llamar, entonces `p` debe añadirse a `o`. Pero si `o` no es extensible, entonces no se pueden definir nuevas propiedades en él.

## 6.4 Borrar propiedades

El operador `delete` ([§4.13.4](#)) elimina una propiedad de un objeto. Su único operando debe ser una expresión de acceso a la propiedad. Sorprendentemente, `delete` no opera sobre el valor de la propiedad sino sobre la propia propiedad:

```
borrar libro.autor; // El objeto libro ahora no tiene la propiedad autor.  
borrar libro["título principal"]; // Ahora tampoco tiene "título principal".
```

El operador de borrado sólo borra las propiedades propias, no las heredadas. (Para eliminar una propiedad heredada, debe eliminarla del objeto prototipo en el que está definida. Hacer esto afecta a todos los objetos que heredan de ese prototipo).

Una expresión de borrado se evalúa como verdadera si el borrado tuvo éxito o si el borrado no tuvo ningún efecto (como el borrado de una propiedad inexistente):

```
let o = {x: 1}; // o tiene la propiedad propia x y hereda la propiedad toString delete  
o.x // => true: borra la propiedad x delete o.x // => true: no hace nada (x no existe)  
pero si true delete o.toString // => true: no hace nada (toString no es una propiedad  
propia)  
delete 1 // => true: sin sentido, pero verdadero de todos modos
```

delete no elimina las propiedades que tienen un atributo *configurable* de false. Algunas propiedades de los objetos incorporados no son configurables, al igual que las propiedades del objeto global creado por la declaración de variables y la declaración de funciones. En modo estricto, el intento de eliminar una propiedad no configurable provoca un TypeError. En modo no estricto, borrar simplemente se evalúa como falso en este caso:

```
// En modo estricto, todos estos borrados lanzan TypeError en lugar de  
devolver false delete Object.prototype // => false: la propiedad no es  
configurable var x = 1; // Declarar una variable global delete globalThis.x  
// => false: no se puede borrar esta propiedad function f() {} //  
Declarar una función global delete globalThis.f // => false: tampoco se  
puede borrar esta propiedad
```

Cuando se borran propiedades configurables del objeto global en modo no estricto, se puede omitir la referencia al objeto global y simplemente seguir el operador de borrado con el nombre de la propiedad:

```
globalThis.x = 1; // Crear una propiedad global configurable (sin let ni var)  
delete x //=> true: esta propiedad se puede borrar
```

En el modo estricto, sin embargo, delete genera un SyntaxError si su operando es un identificador no cualificado como x, y tienes que ser explícito sobre el acceso a la propiedad:

```
delete x; // SyntaxError en modo estricto delete globalThis.x; // Esto funciona
```

## 6.5 Comprobación de las propiedades

Los objetos de JavaScript pueden considerarse conjuntos de propiedades, y a menudo es útil poder comprobar la pertenencia al conjunto, es decir, comprobar si un objeto tiene una propiedad con un nombre determinado. Esto se puede hacer con el operador in, con las funciones hasOwnProperty() y propertyIsEnumerable(), o simplemente consultando la propiedad. Todos los ejemplos mostrados aquí utilizan cadenas como nombres de propiedades, pero también funcionan con Symbols ([§6.10.3](#)).

El operador in espera un nombre de propiedad a su izquierda y un objeto a su derecha. Devuelve true si el objeto tiene una propiedad propia o una propiedad heredada con ese nombre:

```
dejemos que o = { x: 1 };  
"x" in o //=> true: o tiene una propiedad propia "x"  
"y" in o //=> false: o no tiene una propiedad "y"  
"toString" in o //=> true: o hereda una propiedad toString
```

El método hasOwnProperty() de un objeto comprueba si ese objeto tiene una propiedad propia con el nombre dado. Devuelve false para las propiedades heredadas:

```
dejemos que o = { x: 1 };
```

- o. `hasOwnProperty("x")` // => true: o tiene una propiedad propia x
- o. `hasOwnProperty("y")` // => false: o no tiene una propiedad y
- o. `hasOwnProperty("toString")` // => false: `toString` es una propiedad heredada

La función `propertyIsEnumerable()` refina la prueba `hasOwnProperty()`. Sólo devuelve true si la propiedad nombrada es una propiedad propia y su atributo enumerable es true.

Algunas propiedades incorporadas no son enumerables. Las propiedades creadas por el código JavaScript normal son enumerables a menos que se haya utilizado una de las técnicas mostradas en §14.1 para hacerlas no enumerables.

- dejemos que** `o = { x: 1 };`
- o. `propertyIsEnumerable("x")` // => true: o tiene una propiedad propia enumerable x
  - o. `propertyIsEnumerable("toString")` // => false: no es una propiedad propia Object.prototype.propertyIsEnumerable("toString") // => false: no es enumerable

En lugar de utilizar el operador `in`, a menudo es suficiente con consultar la propiedad y utilizar `!==` para asegurarse de que no está indefinida:

- dejemos que** `o = { x: 1 };`
- o. `x !== undefined` // => true: o tiene una propiedad x
  - o. `y !== undefined` // => false: o no tiene una propiedad y
  - o. `toString !== undefined` // => true: o hereda una propiedad `toString`

Hay una cosa que el operador `in` puede hacer que la simple técnica de acceso a propiedades mostrada aquí no puede hacer. `in` puede distinguir entre propiedades que no existen y propiedades que existen pero que han sido establecidas como indefinidas.

Considere este código:

```
let o = { x: undefined }; // La propiedad se establece explícitamente como undefined
o.x !== undefined // => false: la propiedad existe pero no está definida
o.y !== undefined // => false: la propiedad no existe
```

```
"x" en o //=> true: la propiedad existe  
"y" en o //=> false: la propiedad no existe delete o.x; // Eliminar la propiedad x  
"x" en o //=> false: ya no existe
```

## 6.6 Enumeración de propiedades

En lugar de comprobar la existencia de propiedades individuales, a veces queremos iterar u obtener una lista de todas las propiedades de un objeto. Hay varias formas de hacerlo.

El bucle `for/in` se trató en §5.4.5. Ejecuta el cuerpo del bucle una vez por cada propiedad enumerable (propia o heredada) del objeto especificado, asignando el nombre de la propiedad a la variable del bucle. Los métodos incorporados que los objetos heredan no son enumerables, pero las propiedades que su código añade a los objetos son enumerables por defecto. Por ejemplo:

```
let o = {x: 1, y: 2, z: 3}; // Tres propiedades propias enumerables  
o.propertyIsEnumerable("toString") //=> false: no es enumerable for(let p in o) {  
  // Recorre las propiedades console.log(p); // Imprime x, y y z, pero no toString }
```

Para evitar la enumeración de propiedades heredadas con `for/in`, puede añadir una comprobación explícita dentro del cuerpo del bucle:

```
for(let p in o) { if (!o.hasOwnProperty(p)) continue; // Omite las  
propiedades heredadas }  
  
for(let p in o) { if (typeof o[p] === "function") continue; // Saltar todos los  
métodos  
}
```

Como alternativa al uso de un bucle `for/in`, a menudo es más fácil obtener una matriz de nombres de propiedades para un objeto y

luego recorrer esa matriz con un bucle for/of. Hay cuatro funciones que puedes utilizar para obtener un array de nombres de propiedades:

- `Object.keys()` devuelve una matriz con los nombres de las propiedades propias enumerables de un objeto. No incluye las propiedades no enumerables, las propiedades heredadas o las propiedades cuyo nombre es un Símbolo (véase [§6.10.3](#)).
- `Object.getOwnPropertyNames()` funciona como `Object.keys()` pero devuelve también una matriz con los nombres de las propiedades propias no numerables, siempre que sus nombres sean cadenas.
- `Object.getOwnPropertySymbols()` devuelve own propiedades cuyos nombres son Símbolos, sean o no enumerables.
- `Reflect.ownKeys()` devuelve todos los nombres de propiedades propias, tanto enumerables como no enumerables, y tanto de cadena como de símbolo. (Véase [§14.6](#).)

Hay ejemplos del uso de `Object.keys()` con un bucle for/of en [§6.7](#).

### 6.6.1 Orden de enumeración de las propiedades

ES6 define formalmente el orden en que se enumeran las propiedades propias de un objeto. `Object.keys()`, `Object.getOwnPropertyNames()`, `Object.getOwnPropertySymbols()`, `Reflect.ownKeys()`, y otros métodos relacionados como `JSON.stringify()` listan todas las propiedades en el siguiente orden, sujeto a sus propias restricciones adicionales sobre si listan

propiedades no numerables o propiedades cuyos nombres son cadenas o símbolos:

- Las propiedades de las cadenas cuyos nombres son enteros no negativos se enumeran primero, en orden numérico de menor a mayor. Esta regla significa que las matrices y los objetos tipo matriz tendrán sus propiedades enumeradas en orden.
- Después de enumerar todas las propiedades que parecen índices de matrices, se enumeran todas las propiedades restantes con nombres de cadena (incluidas las propiedades que parecen números negativos o números de punto flotante). Estas propiedades se enumeran en el orden en que se añadieron al objeto. Para las propiedades definidas en un literal de objeto, este orden es el mismo en el que aparecen en el literal.
- Por último, las propiedades cuyos nombres son objetos Symbol se enumeran en el orden en que se añadieron al objeto.

El orden de enumeración para el bucle for/in no está tan especificado como para estas funciones de enumeración, pero las implementaciones suelen enumerar las propiedades propias en el orden que se acaba de describir, y luego recorren la cadena de prototipos enumerando las propiedades en el mismo orden para cada objeto prototipo. Tenga en cuenta, sin embargo, que una propiedad no será enumerada si una propiedad con ese mismo nombre ya ha sido enumerada, o incluso si una propiedad no enumerable con el mismo nombre ya ha sido considerada.

## 6.7 Ampliación de objetos

Una operación común en los programas de JavaScript es la necesidad de copiar las propiedades de un objeto a otro objeto. Es fácil hacerlo con un código como este:

```
let target = {x: 1}, source = {y: 2, z: 3}; for(let key of Object.  
keys(source)) { target[key] = source[key];  
} target //=> {x: 1, y: 2, z: 3}
```

Pero como se trata de una operación común, varios frameworks de JavaScript han definido funciones de utilidad, a menudo denominadas `extend()`, para realizar esta operación de copia. Finalmente, en ES6, esta capacidad llega al núcleo del lenguaje JavaScript en forma de `Object.assign()`.

`Object.assign()` espera dos o más objetos como argumentos. Modifica y devuelve el primer argumento, que es el objeto de destino, pero no altera el segundo ni los argumentos posteriores, que son los objetos de origen. Para cada objeto fuente, copia las propiedades propias enumerables de ese objeto (incluyendo aquellas cuyos nombres son Símbolos) en el objeto destino. Procesa los objetos fuente en el orden de la lista de argumentos, de modo que las propiedades del primer objeto fuente anulan las propiedades con el mismo nombre en el objeto destino y las propiedades del segundo objeto fuente (si lo hay) anulan las propiedades con el mismo nombre en el primer objeto fuente.

`Object.assign()` copia las propiedades con operaciones ordinarias de obtención y establecimiento de propiedades, por lo que si un objeto origen tiene un método getter o el objeto destino tiene un

método setter, serán invocados durante la copia, pero no se copiarán ellos mismos.

Una razón para asignar propiedades de un objeto a otro es cuando se tiene un objeto que define valores por defecto para muchas propiedades y se quiere copiar esas propiedades por defecto en otro objeto si no existe ya una propiedad con ese nombre en ese objeto. Usar `Object.assign()` de forma ingenua no hará lo que quieras:

```
Object.assign(o, defaults); // sobrescribe todo lo que hay en o con los defaults
```

En su lugar, lo que puedes hacer es crear un nuevo objeto, copiar los valores por defecto en él, y luego anular esos valores por defecto con las propiedades en `o`:

```
o = Object.assign({}, defaults, o);
```

Veremos en [§6.10.4](#) que también se puede expresar esta operación de copia-anulación de objetos utilizando el operador de propagación ... de esta manera:

```
o = {... predeterminados, ... o};
```

También podríamos evitar la sobrecarga de la creación y copia extra de objetos escribiendo una versión de `Object.assign()` que copie las propiedades sólo si faltan:

```
// Como Object.assign() pero no anula las propiedades existentes // (y
// tampoco maneja las propiedades Symbol) function merge(target, ...
sources) { for(let source of sources) { for(let key of Object.keys(source)) {
    if (!(key in target)) { // Esto es diferente a Object.assign() target[key] =
source[key]; }
} } return target; }

Object.assign({x: 1}, {x: 2, y: 2}, {y: 3, z: 4}) // => {x: 2, y: 3, z: 4} merge({x: 1}, {x: 2, y:
2}, {y: 3, z: 4}) // => {x:
1, y: 2, z: 4}
```

Es sencillo escribir otras utilidades de manipulación de propiedades como esta función merge(). Una función restrict() podría eliminar las propiedades de un objeto si no aparecen en otro objeto de la plantilla, por ejemplo. O una función subtract() podría eliminar todas las propiedades de un objeto de otro objeto.

## 6.8 Serialización de objetos

*La serialización de objetos* es el proceso de convertir el estado de un objeto en una cadena a partir de la cual se puede restaurar posteriormente. Las funciones

JSON.stringify() y JSON.parse() serializan y restauran objetos JavaScript. Estas funciones utilizan el formato de intercambio de datos JSON. JSON significa "JavaScript Object Notation", y su sintaxis es muy similar a la de los literales de objetos y arrays de JavaScript:

```
dejemos que o = {x: 1, y: {z: [false, null, ""]} }; // Definir un objeto de prueba let s
= JSON.stringify(o); // s == '{"x":1, "y": {"z": [false, null, ""]}}' let p = JSON.parse(s);
// p == {x: 1, y: {z: [false, null, ""]}}
```

La sintaxis JSON es un *subconjunto* de la sintaxis de JavaScript, y no puede representar todos los valores de JavaScript. Los objetos, las matrices, las cadenas, los números finitos, true, false y null son compatibles y pueden ser serializados y restaurados. NaN, Infinity y -Infinity se serializan a null. Los objetos de fecha se serializan como cadenas de fecha con formato ISO (véase la sección Date.toJSON()), pero JSON.parse() los deja en forma de cadena y no restaura el objeto Date original. Los objetos Function, RegExp y Error y el valor indefinido no pueden ser serializados o restaurados. JSON.stringify() sólo serializa las propiedades propias enumerables de un objeto. Si el valor de una propiedad no puede ser serializado, esa propiedad es simplemente omitida de la salida stringificada.

Tanto JSON.stringify() como JSON.parse() aceptan la opción segundos argumentos que pueden utilizarse para personalizar el proceso de serialización y/o restauración especificando una lista de propiedades a serializar, por ejemplo, o convirtiendo ciertos valores durante el proceso de serialización o encadenamiento. La documentación completa de estas funciones se encuentra en

## §11.6.

## 6.9 Métodos de los objetos

Como se ha comentado anteriormente, todos los objetos de JavaScript (excepto los creados explícitamente sin prototipo) heredan propiedades de Object.prototype. Estas propiedades heredadas son principalmente métodos, y como están disponibles universalmente, son de particular interés para los programadores de JavaScript. Ya hemos visto las propiedades

hasOwnProperty() y propertyIsEnumerable(), por ejemplo. (Y también hemos cubierto ya bastantes funciones estáticas definidas en el constructor Object, como Object.create() y Object.keys()). Esta sección explica un puñado de métodos universales de objetos que se definen en Object.prototype, pero que están pensados para ser sustituidos por otras implementaciones más especializadas. En las secciones que siguen, mostramos ejemplos de definición de estos métodos en un solo objeto. En el [Capítulo 9](#), aprenderás a definir estos métodos de forma más general para toda una clase de objetos.

### 6.9.1 El método `toString()`

El método `toString()` no toma argumentos; devuelve una cadena que representa de alguna manera el valor del objeto sobre el que se invoca. JavaScript invoca este método de un objeto siempre que necesita convertir el objeto en una cadena. Esto ocurre, por ejemplo, cuando se utiliza el operador `+` para concatenar una cadena con un objeto o cuando se pasa un objeto a un método que espera una cadena.

El método `toString()` por defecto no es muy informativo (aunque es útil para determinar la clase de un objeto, como veremos en [§14.4.3](#)).

Por ejemplo, la siguiente línea de código simplemente se evalúa a la cadena "[objeto Objeto)": `let s = { x: 1, y: 1 }.toString(); // s == "[objeto  
Objeto]"`

Como este método por defecto no muestra mucha información útil, muchas clases definen sus propias versiones de `toString()`. Por

ejemplo, cuando un array se convierte en una cadena, se obtiene una lista de los elementos del array, cada uno de ellos convertido en una cadena, y cuando una función se convierte en una cadena, se obtiene el código fuente de la función. Puedes definir tu propio método `toString()` así:

```
let punto = { x: 1,  
y: 2,  
    toString: function() { return `(${this.x}, ${this.y})`; } };  
String(punto) //=> "(1, 2)": toString() se utiliza para la conversión de cadenas
```

## 6.9.2 El método `toLocaleString()`

Además del método básico `toString()`, todos los objetos tienen un método `toLocaleString()`. El propósito de este método es devolver una representación de cadena localizada del objeto. El método por defecto `toLocaleString()` definido por `Object` no realiza ninguna localización por sí mismo: simplemente llama a `toString()` y devuelve ese valor.

Las clases `Date` y `Number` definen versiones personalizadas de `toLocaleString()` que intentan formatear los números, las fechas y las horas según las convenciones locales. `Array` define un método `toLocaleString()` que funciona como `toString()`, excepto que formatea los elementos del array llamando a sus métodos `toLocaleString()` en lugar de a sus métodos `toString()`. Se puede hacer lo mismo con un objeto `punto` como este:

```
let point = { x: 1000,
  y: 2000,
  toString: function() { return `(${this.x}, ${this.y})`; },
  toLocaleString: function() {
    return `(${this.x.toLocaleString()}, ${this.y.toLocaleString()})`;
  }
}; point.toString() //=> "(1000, 2000)" point.toLocaleString() //=>
"(1,000, 2,000)": observe los separadores de miles
```

Las clases de internacionalización documentadas en [§11.7](#) pueden ser útiles al implementar un método `toLocaleString()`.

### 6.9.3 El método `valueOf()`

El método `valueOf()` es muy parecido al método `toString()`, pero se llama cuando JavaScript necesita convertir un objeto en algún tipo primitivo que no sea una cadena, normalmente un número.

JavaScript llama a este método automáticamente si un objeto se utiliza en un contexto en el que se requiere un valor primitivo. El método `valueOf()` por defecto no hace nada interesante, pero algunas de las clases incorporadas definen su propio método `valueOf()`. La clase `Date` define `valueOf()` para convertir fechas en números, y esto permite que los objetos `Date` sean comparados cronológicamente con `<y>`. Se podría hacer algo similar con un objeto punto, definiendo un método `valueOf()` que devuelva la distancia desde el origen al punto:

```
let punto = { x: 3, y: 4, valueOf: function() { return Math.hypot(this.x, this.y); }
};
Number(point) //=> 5: valueOf() se utiliza para las conversiones a números
point > 4 //=> true point > 5 //=> false point < 6 //=> true
```

## 6.9.4 El método toJSON()

Object.prototype no define un método toJSON(), pero el método JSON.stringify() (ver §6.8) busca un método toJSON() en cualquier objeto que se le pida serializar. Si este método existe en el objeto a serializar, se invoca, y el valor de retorno se serializa, en lugar del objeto original. La clase Date (§11.4) define un método toJSON() que devuelve una representación de cadena serializable de la fecha. Podríamos hacer lo mismo para nuestro objeto Point de esta manera:

```
let punto = { x: 1,  
y: 2,  
    toString: function() { return `(${this.x}, ${this.y})`;  
},  
  
    toJSON: function() { return this.toString(); } };  
JSON.stringify([punto]) // => '["(1, 2)"]'
```

## 6.10 Sintaxis literal de objetos extendida

Las versiones recientes de JavaScript han ampliado la sintaxis de los literales de objeto de varias maneras útiles. Las siguientes subsecciones explican estas extensiones.

### 6.10.1 Propiedades abreviadas

Supongamos que tienes valores almacenados en las variables x e y y quieras crear un objeto con propiedades llamadas x e y que contengan esos valores. Con la sintaxis básica de los literales de objeto, terminarías repitiendo cada identificador dos veces:

```
dejemos que x = 1,  
y = 2; dejemos que  
o = { x: x, y: y };
```

En ES6 y posteriores, puedes eliminar los dos puntos y una copia del identificador y terminar con un código mucho más simple:

```
dejemos que x = 1, y = 2; dejemos que o = { x, y };
o. x + o. y // => 3
```

## 6.10.2 Nombres de propiedades computados

A veces necesitas crear un objeto con una propiedad específica, pero el nombre de esa propiedad no es una constante en tiempo de compilación que puedas escribir literalmente en tu código fuente. En su lugar, el nombre de la propiedad que necesitas está almacenado en una variable o es el valor de retorno de una función que invocas. No puedes utilizar un literal de objeto básico para este tipo de propiedades. En su lugar, tienes que crear un objeto y luego añadir las propiedades deseadas como un paso extra:

```
const PROPERTY_NAME = "p1"; function computePropertyName() {
  return "p" + 2;
}

let o = {};
o[PROPERTY_NAME] = 1;
o[computePropertyName()] = 2;
```

Es mucho más sencillo configurar un objeto como éste con una función de ES6 conocida como *propiedades computadas* que permite tomar los corchetes del código anterior y trasladarlos directamente al literal del objeto:

```
const PROPERTY_NAME = "p1"; function computePropertyName() {  
    return "p" + 2; }  
  
let p = { [PROPERTY_NAME]: 1,  
          [computePropertyName()]: 2 };  
  
p. p1 + p. p2 //=> 3
```

Con esta nueva sintaxis, los corchetes delimitan una expresión JavaScript arbitraria. Esta expresión se evalúa, y el valor resultante (convertido a una cadena, si es necesario) se utiliza como nombre de la propiedad.

Una situación en la que podrías querer usar propiedades computadas es cuando tienes una biblioteca de código JavaScript que espera que se le pasen objetos con un conjunto particular de propiedades, y los nombres de esas propiedades están definidos como constantes en esa biblioteca. Si estás escribiendo código para crear los objetos que se pasarán a esa biblioteca, podrías codificar los nombres de las propiedades, pero te arriesgarías a tener errores si escribes el nombre de la propiedad de forma incorrecta en cualquier lugar, y te arriesgarías a tener problemas de desajuste de versiones si una nueva versión de la biblioteca cambia los nombres de las propiedades requeridas. En su lugar, puede encontrar que hace que su código sea más robusto para utilizar la sintaxis de la propiedad calculada con las constantes de nombres de propiedades definidas por la biblioteca.

### 6.10.3 Símbolos como nombres de propiedades

La sintaxis de la propiedad computada permite otra característica literal de objeto muy importante. En ES6 y posteriores, los

•

nombres de las propiedades pueden ser cadenas o símbolos. Si asignas un símbolo a una variable o constante, entonces puedes usar ese símbolo como nombre de propiedad usando la sintaxis de propiedad computada:

```
const extension = Symbol("mi símbolo de extensión"); let o = { [extension]: { /* datos de extensión almacenados en este objeto */ } }; o[extensión].x = 0; // Esto no entrará en conflicto con otras propiedades de o
```

Como se explica en §3.6, los símbolos son valores opacos. No se puede hacer nada con ellos más que utilizarlos como nombres de propiedades. Sin embargo, cada Symbol es diferente de cualquier otro Symbol, lo que significa que los Symbols son buenos para crear nombres de propiedades únicos. Cree un nuevo Symbol llamando a la función de fábrica Symbol(). (Los Symbols son valores primitivos, no objetos, por lo que Symbol() no es una función constructora que se invoca con new). El valor devuelto por Symbol() no es igual a ningún otro Symbol u otro valor. Puede pasar una cadena a Symbol(), y esta cadena se utiliza cuando su Symbol se convierte en una cadena. Pero esto es sólo una ayuda para la depuración: dos Symbols creados con el mismo argumento de cadena siguen siendo diferentes entre sí.

El objetivo de Symbols no es la seguridad, sino definir un mecanismo de extensión seguro para los objetos de JavaScript. Si obtienes un objeto de un código de terceros que no controlas y necesitas añadir algunas de tus propias propiedades a ese objeto, pero quieres estar seguro de que tus propiedades no entrarán en conflicto con ninguna propiedad que pueda existir ya en el objeto, puedes utilizar de forma segura Symbols como nombres de

propiedades. Si hace esto, también puede estar seguro de que el código de terceros no alterará accidentalmente sus propiedades con nombre de símbolo. (Ese código de terceros podría, por supuesto, usar `Object.getOwnPropertySymbols()` para descubrir los Symbols que estás usando y podría entonces alterar o borrar tus propiedades. Por eso los Symbols no son un mecanismo de seguridad).

#### 6.10.4 Operador de difusión

En ES2018 y posteriores, puedes copiar las propiedades de un objeto existente en un nuevo objeto utilizando el "operador de propagación" ... dentro de un literal de objeto:

```
let posición = { x: 0, y: 0 }; let dimensions = { width: 100, height: 75 };
let rect = { ... posición, ... dimensions }; rect.x + rect.y + rect.width +
rect.height // => 175
```

En este código, las propiedades de los objetos `position` y `dimensions` se "extienden" en el literal del objeto `rect` como si se hubieran escrito literalmente dentro de esas llaves. Tenga en cuenta que esta sintaxis ... se llama a menudo operador de propagación, pero no es un verdadero operador de JavaScript en ningún sentido. En su lugar, es una sintaxis de caso especial disponible sólo dentro de los literales de objeto. (Los tres puntos se utilizan con otros fines en otros contextos de JavaScript, pero los literales de objeto son el único contexto en el que los tres puntos provocan este tipo de interpolación de un objeto en otro).

Si el objeto que se propaga y el objeto en el que se propaga tienen ambos una propiedad con el mismo nombre, el valor de esa propiedad será el último:

```
let o = { x: 1 }; let p = { x: 0, ... o };
p.x //=> 1: el valor del objeto o anula el valor inicial let q = { ... o, x: 2 };
q.x //=> 2: el valor 2 anula el valor anterior de o.
```

Tenga en cuenta también que el operador de propagación sólo propaga las propiedades propias de un objeto, no las heredadas:

```
let o = Object.create({x: 1}); // o hereda la propiedad x let p = { ... o };
p.x //=> indefinido
```

Por último, vale la pena señalar que, aunque el operador de propagación es sólo tres pequeños puntos en su código, puede representar una cantidad sustancial de trabajo para el intérprete de JavaScript. Si un objeto tiene  $n$  propiedades, el proceso de propagación de esas propiedades en otro objeto es probablemente una operación  $O(n)$ . Esto significa que si te encuentras usando ... dentro de un bucle o función recursiva como una forma de acumular datos en un objeto grande, puedes estar escribiendo un algoritmo  $O(n^2)$  ineficiente que no escalará bien a medida que  $n$  se haga más grande.

## 6.10.5 Métodos abreviados

Cuando una función se define como una propiedad de un objeto, llamamos a esa función un *método* (tendremos mucho más que decir sobre los métodos en los capítulos [8](#) y [9](#)). Antes de ES6, se definía un método en un literal de objeto utilizando una expresión de definición de función, al igual que se definía cualquier otra propiedad de un objeto:

```
let square = { area: function() { return this.side * this.side; }, side: 10 };
square.area() //=> 100
```

En ES6, sin embargo, la sintaxis de los literales de objetos (y también la sintaxis de definición de clases que veremos en [el capítulo 9](#)) se ha ampliado para permitir un atajo en el que se omiten la palabra clave de función y los dos puntos, lo que da lugar a un código como éste:

```
let square = { area() { return this.side * this.side; }, side: 10 };
square.area() //=> 100
```

Ambas formas del código son equivalentes: ambas añaden una propiedad llamada `area` al objeto literal, y ambas establecen el valor de esa propiedad a la función especificada. La sintaxis abreviada aclara que `area()` es un método y no una propiedad de datos como `side`.

Cuando se escribe un método utilizando esta sintaxis abreviada, el nombre de la propiedad puede adoptar cualquiera de las formas que son legales en un literal de objeto: además de un identificador JavaScript normal como el área de nombres anterior, también se pueden utilizar literales de cadena y nombres de propiedades computados, que pueden incluir nombres de propiedades `Symbol`:

```
const MÉTODO_NOMBRE = "m"; const símbolo = Symbol();
let weirdMethods = { "método Con Espacios"(x) { retorno x +
1; },
[MÉTODO_NOMBRE](x) { devuelve x + 2; },
[símbolo](x) { devuelve x + 3; }; weirdMethods["método
con espacios"](1) //=> 2 weirdMethods[METHOD_NAME](1)
}; weirdMethods["método Con Espacios"](1) //=> 2
weirdMethods[METHOD_NAME](1) //=> 3
weirdMethods[símbolo](1) //=> 4
```

Utilizar un `Símbolo` como nombre de método no es tan extraño como parece. Para hacer que un objeto sea iterable (para poder

utilizarlo con un bucle `for/of`), debes definir un método con el nombre simbólico `Symbol.iterator`, y hay ejemplos de cómo hacer exactamente eso en [el capítulo 12](#).

### 6.10.6 Obtención y fijación de propiedades

Todas las propiedades de objeto que hemos discutido hasta ahora en este capítulo han sido *propiedades de datos* con un nombre y un valor ordinario. JavaScript también admite *propiedades accesorias*, que no tienen un único valor sino que tienen uno o dos métodos de acceso: un *getter* y/o un *setter*.

Cuando un programa consulta el valor de una propiedad accesoria, JavaScript invoca el método *getter* (sin pasar argumentos). El valor de retorno de este método se convierte en el valor de la expresión de acceso a la propiedad. Cuando un programa establece el valor de una propiedad accesoria, JavaScript invoca el método *setter*, pasando el valor del lado derecho de la asignación. Este método es el responsable de "establecer", en cierto sentido, el valor de la propiedad. El valor de retorno del método *setter* se ignora.

Si una propiedad tiene un método *getter* y un método *setter*, es una propiedad de lectura/escritura. Si sólo tiene un método *getter*, es una propiedad de sólo lectura. Y si sólo tiene un método *setter*, es una propiedad de sólo escritura (algo que no es posible con las propiedades de datos), y los intentos de leerla siempre se evalúan como indefinidos.

Las propiedades accesorias pueden definirse con una extensión de la sintaxis literal del objeto (a diferencia de las otras extensiones

de ES6 que hemos visto aquí, los getters y setters se introdujeron en ES5):

```
let o = { // Una propiedad de datos ordinaria
  dataProp: valor,
  // Una propiedad accesoria definida como un par de funciones.
  get accessorProp() { return this.dataProp; },
  set accessorProp(value) { this.dataProp = value; }
};
```

Las propiedades accesorias se definen como uno o dos métodos cuyo nombre es el mismo que el de la propiedad. Se parecen a los métodos ordinarios definidos con la abreviatura de ES6, salvo que las definiciones de getter y setter llevan el prefijo get o set. (En ES6, también se pueden utilizar nombres de propiedades computados al definir getters y setters. Simplemente reemplaza el nombre de la propiedad después de get o set con una expresión entre corchetes).

Los métodos de acceso definidos anteriormente simplemente obtienen y establecen el valor de una propiedad de datos, y no hay ninguna razón para preferir la propiedad de acceso sobre la propiedad de datos. Pero como ejemplo más interesante, considere el siguiente objeto que representa un punto cartesiano 2D. Tiene propiedades de datos ordinarias para representar las coordenadas  $x$  e  $y$  del punto, y tiene propiedades accesorias que dan las coordenadas polares equivalentes del punto:

```

let p = { // x e y son propiedades de datos regulares de lectura y escritura.
  x: 1.0, y: 1.0,

  // r es una propiedad accesoria de lectura-escritura con getter y setter. // No
  // olvides poner una coma después de los métodos de acceso.
  get r() { return Math. hypot(this. x, this. y); }, set r(newvalue) { let
  oldvalue = Math. hypot(this. x, this. y); let ratio = newvalue/oldvalue;
  this. x *= ratio; this. y *= ratio; },

  // theta es una propiedad de acceso de sólo lectura con getter solamente.
  get theta() { return Math. atan2(this. y, this. x); } };

p. r //=> Math.SQRT2
p. theta //=> Math.PI / 4

```

Observe el uso de la palabra clave `this` en los getters y setters de este ejemplo. JavaScript invoca estas funciones como métodos del objeto sobre el que se definen, lo que significa que dentro del cuerpo de la función, `this` se refiere al objeto punto `p`. Así, el método getter de la propiedad `r` puede referirse a las propiedades `x` e `y` como `this.x` y `this.y`. Los métodos y la palabra clave `this` se tratan con más detalle en [§8.2.2](#).

Las propiedades accesorias se heredan, al igual que las propiedades de datos, por lo que puedes utilizar el objeto `p` definido anteriormente como prototipo para otros puntos. Puedes dar a los nuevos objetos sus propias propiedades `x` e `y`, y heredarán las propiedades `r` y `theta`:

```

let q = Object. create(p); // Un nuevo objeto que hereda getters y setters
q. x = 3; q. y = 4; // Crear las propiedades de datos propias de q

```

```
q. r //=> 5: las propiedades accesorias heredadas funcionan  
q. theta //=> Math.atan2(4, 3)
```

El código anterior utiliza propiedades accesorias para definir una API que proporciona dos representaciones (coordenadas cartesianas y coordenadas polares) de un único conjunto de datos. Otras razones para utilizar propiedades accesorias incluyen la comprobación de la sanidad de las escrituras de propiedades y la devolución de valores diferentes en cada lectura de propiedades:

```
// Este objeto genera números de serie estrictamente crecientes const serialnum = {  
    // Esta propiedad de datos contiene el siguiente número de serie. // El _ en  
    // el nombre de la propiedad indica que es de uso interno. _n: 0,
```

```
    // Devuelve el valor actual y lo incrementa get next() { return this;  
    _n++; },
```

```
// Establece un nuevo valor de n, pero sólo si es mayor que el actual const  
next(n) { if (n > this._n) this._n = n;  
    else throw new Error("el número de serie sólo puede establecerse en un  
valor mayor"); }; serialnum.next = 10; // Establece el número de serie inicial. ;  
serialnum.next = 10; // Establecer el número de serie inicial serialnum.next //  
=> 10 serialnum.next // => 11: valor diferente cada vez que obtenemos next
```

Por último, he aquí un ejemplo más que utiliza un método getter para implementar una propiedad con un comportamiento "mágico":

```
// Este objeto tiene propiedades accesorias que devuelven números aleatorios.  
// La expresión "random.octet", por ejemplo, produce un número aleatorio // entre 0  
y 255 cada vez que se evalúa. const random = { get octet() { return Math.  
floor(Math.random()*256); }, get uint16() { return Math.floor(Math.  
random()*65536); }, get int16() { return Math.floor(Math.random()*65536)-32768;  
}  
};
```

## 6.11 Resumen

En este capítulo se han documentado los objetos de JavaScript con gran detalle, cubriendo temas que incluyen:

- Terminología básica de los objetos, incluyendo el significado de términos como *enumerable* y *propiedad propia*.
- Sintaxis literal de objetos, incluidas las numerosas novedades de ES6 y más tarde.
- Cómo leer, escribir, borrar, enumerar y comprobar la presencia de las propiedades de un objeto.
- Cómo funciona la herencia basada en prototipos en JavaScript y cómo crear un objeto que herede de otro objeto con `Object.create()`.
- Cómo copiar propiedades de un objeto a otro con `Object.assign()`.

Todos los valores de JavaScript que no son valores primitivos son objetos. Esto incluye tanto las matrices como las funciones, que son los temas de los dos próximos capítulos.



# Capítulo 7. Arrays

---

Este capítulo documenta los arrays, un tipo de datos fundamental en JavaScript y en la mayoría de los demás lenguajes de programación. Un *array* es una colección ordenada de valores. Cada valor se llama *elemento*, y cada elemento tiene una posición numérica en el array, conocida como su *índice*. Las matrices de JavaScript *no están tipificadas*: un elemento de la matriz puede ser de cualquier tipo, y diferentes elementos de la misma matriz pueden ser de diferentes tipos. Los elementos de una matriz pueden ser incluso objetos u otras matrices, lo que permite crear estructuras de datos complejas, como matrices de objetos y matrices de matrices. Las matrices de JavaScript *se basan en el cero* y utilizan índices de 32 bits: el índice del primer elemento es 0, y el índice más alto posible es  $4294967294$  ( $2^{32}-2$ ), para un tamaño máximo de matriz de 4.294.967.295 elementos. Las matrices de JavaScript son *dinámicas*: crecen o se reducen según sea necesario, y no es necesario declarar un tamaño fijo para la matriz cuando se crea o reasignarla cuando el tamaño cambia. Las matrices de JavaScript pueden ser *escasas*: los elementos no tienen por qué tener índices contiguos, y puede haber huecos. Todos los arrays de JavaScript tienen una propiedad de longitud. Para las matrices no dispersas, esta propiedad especifica el número de elementos de la matriz. En el caso de las matrices dispersas, la longitud es mayor que el índice más alto de cualquier elemento.

Los arrays de JavaScript son una forma especializada de objeto de JavaScript, y los índices de los arrays son realmente poco más que nombres de propiedades que resultan ser enteros. Hablaremos más sobre las especializaciones de los arrays en otra parte de este capítulo. Las implementaciones suelen optimizar los arrays para que el acceso a los elementos de los arrays indexados numéricamente sea generalmente más rápido que el acceso a las propiedades de los objetos normales.

Los arrays heredan propiedades de `Array.prototype`, que define un rico conjunto de métodos de manipulación de arrays, cubiertos en [§7.8](#). La mayoría de estos métodos son *genéricos*, lo que significa que funcionan correctamente no sólo para arrays reales, sino para cualquier "objeto tipo array". Hablaremos de los objetos tipo array en [§7.9](#). Por último, las cadenas de JavaScript se comportan como arrays de caracteres, y lo discutiremos en [§7.10](#).

ES6 introduce un conjunto de nuevas clases de arrays conocidas colectivamente como "arrays tipados". A diferencia de las matrices normales de JavaScript, las matrices tipadas tienen una longitud fija y un tipo de elemento numérico fijo. Ofrecen un alto rendimiento y un acceso a nivel de bytes a los datos binarios y se tratan en [§11.2](#).

## 7.1 Creación de matrices

Hay varias formas de crear matrices. Las subsecciones siguientes explican cómo crear matrices con:

- Literales de matrices
- El operador ... spread en un objeto iterable
- El constructor Array()
- Los métodos de fábrica Array.of() y Array.from()

### 7.1.1 Literales de matrices

La forma más sencilla de crear una matriz es con un literal de matriz, que es simplemente una lista separada por comas de los elementos de la matriz entre corchetes. Por ejemplo:

```
let empty = []; // Un array sin elementos let primes = [2, 3, 5, 7, 11]; // Un array con 5 elementos numéricos let misc = [ 1.1, true, "a", ]; // 3 elementos de varios tipos + coma final
```

Los valores de un literal de matriz no necesitan ser constantes; pueden ser expresiones arbitrarias:

```
deje base = 1024;  
let table = [base, base+1, base+2, base+3];
```

Los literales de las matrices pueden contener literales de objetos u otros literales de matrices:

```
dejemos b = [[1, {x: 1, y: 2}], [2, {x: 3, y: 4}]];
```

Si un literal de matriz contiene varias comas en una fila, sin ningún valor entre ellas, la matriz es escasa (véase §7.3). Los elementos de la matriz para los que se omiten valores no existen, pero aparecen como indefinidos si se consultan:

```
let count = [1,,3]; // Elementos en los índices 0 y 2. Ningún elemento en el índice 1  
let undefs = [,,]; // Un array sin elementos pero con una longitud de 2
```

La sintaxis de los literales de las matrices permite una coma final opcional, por lo que `[,,]` tiene una longitud de 2, no de 3.

### 7.1.2 El Operador de Propagación

En ES6 y posteriores, se puede utilizar el "operador de propagación", ..., para incluir los elementos de una matriz dentro de un literal de matriz:

```
dejemos a = [1, 2, 3];
let b = [0, ... a, 4]; // b == [0, 1, 2, 3, 4]
```

Los tres puntos "extienden" el array `a` para que sus elementos se conviertan en elementos dentro del literal del array que se está creando. Es como si el `...a` fuera reemplazado por los elementos del array `a`, listados literalmente como parte del literal del array que lo encierra. (Ten en cuenta que, aunque llamamos a estos tres puntos operador de propagación, no es un verdadero operador porque sólo puede usarse en literales de array y, como veremos más adelante en el libro, en invocaciones de funciones).

El operador de propagación es una forma conveniente de crear una copia (superficial) de una matriz:

```
let original = [1,2,3]; let copy = [... original]; copy[0] = 0; // Modificar la
copia no cambia el original[0] //=> 1
```

El operador spread funciona en cualquier objeto iterable. (Los objetos *iterables* son sobre los que itera el bucle `for/of`; los vimos por primera vez en [§5.4.4](#), y veremos mucho más sobre ellos en el [capítulo 12](#)). Las cadenas son iterables, por lo que puede utilizar un

operador de propagación para convertir cualquier cadena en una matriz de cadenas de un solo carácter:

```
let dígitos = [... "0123456789ABCDEF"]; dígitos //=>
["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B", "C", "D", "E",
 "F"]
```

Los objetos Set ([§11.1.1](#)) son iterables, por lo que una forma fácil de eliminar los elementos duplicados de un array es convertir el array en un set y luego volver a convertir el set en un array utilizando el operador spread:

```
let letras = [... "hola mundo"];
[... new Set(letters)] //=> ["h", "e", "l", "o",
", "w", "r", "d"]
```

### 7.1.3 El constructor Array()

Otra forma de crear un array es con el constructor Array(). Puedes invocar este constructor de tres maneras distintas:

- Llámalo sin argumentos:

```
let a = new Array();
```

Este método crea un array vacío sin elementos y es equivalente al literal de array [].

- Llámalo con un único argumento numérico, que especifica una longitud:

```
let a = new Array(10);
```

Esta técnica crea un array con la longitud especificada. Esta forma del constructor Array() puede utilizarse para preasignar una matriz cuando se sabe de antemano cuántos elementos se necesitarán. Tenga en cuenta que

no se almacenan valores en el array, y que las propiedades de índice del array "0", "1", etc. ni siquiera están definidas para el array.

- Especifica explícitamente dos o más elementos de la matriz o un solo elemento no numérico para la matriz:

```
let a = new Array(5, 4, 3, 2, 1,  
"probando, probando");
```

En esta forma, los argumentos del constructor se convierten en los elementos del nuevo array. Utilizar un literal de matriz es casi siempre más sencillo que este uso del constructor Array().

#### 7.1.4 Array.of()

Cuando la función constructora Array() es invocada con un argumento numérico, utiliza ese argumento como longitud de la matriz. Pero cuando se invoca con más de un argumento numérico, trata esos argumentos como elementos de la matriz que se va a crear. Esto significa que el constructor Array() no puede utilizarse para crear una matriz con un solo elemento numérico.

En ES6, la función Array.of() aborda este problema: es un método de fábrica que crea y devuelve un nuevo array, utilizando sus valores de argumento (independientemente de cuántos sean) como los elementos del array:

Array. **of()** //=> []; *devuelve un array vacío sin argumentos* Array. **of(10)** //=> [10];  
*puede crear arrays con un único argumento numérico* Array. **of(1,2,3)** //=> [1, 2, 3]

#### 7.1.5 Array.from()

•

Array.from es otro método de fábrica de arrays introducido en ES6.

Espera un objeto iterable o tipo array como primer argumento y devuelve un nuevo array que contiene los elementos de ese objeto. Con un argumento iterable, Array.from(iterable) funciona como lo hace el operador de propagación [...iterable]. También es una forma sencilla de hacer una copia de un array:

```
let copy = Array. from(original);
```

Array.from() también es importante porque define una forma de hacer una copia de un objeto tipo array. Los objetos tipo array son objetos que no son arrays y que tienen una propiedad de longitud numérica y tienen valores almacenados con propiedades cuyos nombres son enteros. Cuando se trabaja con JavaScript del lado del cliente, los valores de retorno de algunos métodos del navegador web son tipo array, y puede ser más fácil trabajar con ellos si primero los conviertes en verdaderos arrays:

```
let truearray = Array. from(arraylike);
```

Array.from() también acepta un segundo argumento opcional. Si pasas una función como segundo argumento, entonces mientras se construye el nuevo array, cada elemento del objeto fuente se pasará a la función que especifiques, y el valor de retorno de la función se almacenará en el array en lugar del valor original. (Esto es muy parecido al método array map() que se introducirá más adelante en el capítulo, pero es más eficiente realizar el mapeo mientras se construye el array que construir el array y luego mapearlo a otro nuevo array).

## 7.2 Lectura y escritura de elementos de la matriz

Para acceder a un elemento de una matriz se utiliza el operador [].

A la izquierda de los corchetes debe aparecer una referencia a la matriz. Dentro de los corchetes debe aparecer una expresión arbitraria que tenga un valor entero no negativo. Esta sintaxis se puede utilizar tanto para leer como para escribir el valor de un elemento de una matriz. Por lo tanto, las siguientes son todas las declaraciones legales de JavaScript:

```
let a = ["mundo"]; // Empezar con un array de un elemento let value = a[0];
// Leer elemento 0 a[1] = 3.14; // Escribir elemento 1 let i = 2; a[i] = 3; //
Escribir elemento 2 a[i + 1] = "hola"; // Escribir elemento 3 a[a[i]] = a[0]; //
Leer elementos 0 y 2, escribir elemento 3
```

Lo especial de los arrays es que cuando utilizas nombres de propiedades que son enteros no negativos menores que  $2^{32}-1$ , el array mantiene automáticamente el valor de la propiedad length por ti. En lo anterior, por ejemplo, creamos un array a con un solo elemento. Luego asignamos valores en los índices 1, 2 y 3. La propiedad de longitud del array cambió a medida que lo hacíamos, así:

```
a. longitud // => 4
```

Recuerda que los arrays son un tipo de objeto especializado. Los corchetes utilizados para acceder a los elementos del array funcionan igual que los corchetes utilizados para acceder a las propiedades de los objetos. JavaScript convierte el índice numérico de la matriz que se especifica en una cadena -el índice 1 se convierte en la cadena "1"- y luego utiliza esa cadena como nombre de la propiedad. No hay nada especial en la conversión del

índice de un número a una cadena: también se puede hacer con objetos normales:

```
let o = {};  
// Crear un objeto plano o[1] = "uno"; //  
Indexarlo con un entero o["1"] //=> "uno"; los nombres  
de las propiedades numéricas y de cadena son los mismos
```

Es útil distinguir claramente un índice de *matriz* de un nombre de propiedad de *objeto*. Todos los índices son nombres de propiedades, pero sólo los nombres de propiedades que son enteros entre 0 y <sup>2<sup>32</sup>-2</sup> son índices. Todos los arrays son objetos, y puedes crear propiedades con cualquier nombre en ellos. Sin embargo, si utilizas propiedades que son índices de arrays, los arrays tienen el comportamiento especial de actualizar su propiedad de longitud según sea necesario.

Tenga en cuenta que puede indexar una matriz utilizando números que sean negativos o que no sean enteros. Cuando se hace esto, el número se convierte en una cadena, y esa cadena se utiliza como nombre de la propiedad. Dado que el nombre no es un número entero no negativo, se trata como una propiedad de objeto normal, no como un índice de matriz. Además, si se indexa un array con una cadena que resulta ser un entero no negativo, se comporta como un índice de array, no como una propiedad de objeto. Lo mismo ocurre si se utiliza un número de punto flotante que es igual a un entero:

```
a[-1.23] = true;  
// Esto crea una propiedad llamada "-1.23" a["1000"] = 0;  
// Este es el elemento 1001 de la matriz a[1.000] = 1;  
// Índice 1 de la matriz. Igual que a[1] = 1;
```

El hecho de que los índices de los arrays sean simplemente un tipo especial de nombre de propiedad de un objeto significa que los arrays de JavaScript no tienen la noción de un error "fuera de límites". Cuando intentas consultar una propiedad inexistente de cualquier objeto, no obtienes un error; simplemente obtienes `undefined`. Esto es tan cierto para los arrays como para los objetos:

```
let a = [true, false]; // Esta matriz tiene elementos en los índices  
0 y 1  
a[2] //=> indefinido; no hay ningún elemento en este índice.  
a[-1] //=> indefinido; no hay ninguna propiedad con este nombre.
```

## 7.3 Matrices dispersas

Una matriz *dispersa* es aquella en la que los elementos no tienen índices contiguos que comiencen en 0. Normalmente, la propiedad `length` de una matriz especifica el número de elementos de la misma. Si el array es disperso, el valor de la propiedad `length` es mayor que el número de elementos. Los arrays dispersos pueden crearse con el constructor `Array()` o simplemente asignando a un índice de array mayor que la longitud actual del array.

```
let a = new Array(5); // No hay elementos, pero a.length es 5.  
a = []; // Crear un array sin elementos y con longitud = 0.  
a[1000] = 0; // La asignación añade un elemento pero establece la longitud en 1001.
```

Más adelante veremos que también se puede hacer un array disperso con el operador de borrado.

Las matrices que son suficientemente dispersas suelen implementarse de una manera más lenta y eficiente en cuanto a la memoria que las matrices densas, y la búsqueda de elementos en

una matriz de este tipo tomará tanto tiempo como la búsqueda regular de propiedades de objetos.

Tenga en cuenta que cuando se omite un valor en un literal de matriz (utilizando comas repetidas como en [1,,3]), la matriz resultante es escasa, y los elementos omitidos simplemente no existen:

```
let a1 = [,]; // Esta matriz no tiene elementos y tiene una longitud de 1 let a2 =  
[undefined]; // Esta matriz tiene un elemento indefinido  
0 en a1 //=> falso: a1 no tiene ningún elemento con índice 0  
0 en a2 //=> true: a2 tiene el valor indefinido en el índice 0
```

Entender los arrays dispersos es una parte importante para comprender la verdadera naturaleza de los arrays de JavaScript. Sin embargo, en la práctica, la mayoría de los arrays de JavaScript con los que se trabaja no son dispersos. Y, si tienes que trabajar con un array disperso, tu código probablemente lo tratará igual que trataría un array no disperso con elementos indefinidos.

## 7.4 Longitud de la matriz

Todos los arrays tienen una propiedad de longitud, y es esta propiedad la que diferencia a los arrays de los objetos normales de JavaScript. Para los arrays que son densos (es decir, no dispersos), la propiedad `length` especifica el número de elementos del array. Su valor es uno más que el índice más alto de la matriz:

```
[].length //=> 0: el array no tiene elementos  
["a", "b", "c"].length //=> 3: el índice más alto es 2, la longitud es 3
```

Cuando un array es disperso, la propiedad de longitud es mayor que el número de elementos, y todo lo que podemos decir al respecto es que se garantiza que la longitud es mayor que el índice de cada elemento del array. O, dicho de otro modo, un array (disperso o no) nunca tendrá un elemento cuyo índice sea mayor o igual que su longitud. Para mantener esta invariante, los arrays tienen dos comportamientos especiales. El primero lo hemos descrito anteriormente: si se asigna un valor a un elemento del array cuyo índice *i* es mayor o igual que la longitud actual del array, el valor de la propiedad *length* se establece en *i+1*.

El segundo comportamiento especial que implementan los arrays para mantener la invariante de longitud es que, si se establece la propiedad de longitud a un entero no negativo *n* menor que su valor actual, cualquier elemento del array cuyo índice sea mayor o igual que *n* se elimina del array:

```
a = [1,2,3,4,5]; // Comienza con un array de 5 elementos.  
a.longitud = 3; // a es ahora [1,2,3].  
a.longitud = 0; // Eliminar todos los elementos. a es [].  
a.longitud = 5; // La longitud es 5, pero no hay elementos, como un nuevo Array(5)
```

También puedes establecer la propiedad de longitud de un array a un valor mayor que su valor actual. Al hacer esto no se añade ningún elemento nuevo al array; simplemente se crea un área escasa al final del array.

## 7.5 Añadir y eliminar elementos de la matriz

Ya hemos visto la forma más sencilla de añadir elementos a un array: basta con asignar valores a los nuevos índices:

```
let a = []; // Empezar con un array vacío. a[0] = "cero"; // Y  
añadirle elementos. a[1] = "uno";
```

También puedes utilizar el método `push()` para añadir uno o más valores al final de un array:

```
let a = []; // Empieza con un array vacío  
a.push("cero"); // Añadir un valor al final. a = ["cero"]  
a.push("uno", "dos"); // Añadir dos valores más. a = ["cero", "uno", "dos"]
```

Empujar un valor a un array a es lo mismo que asignar el valor `a[a.length]`. Puede utilizar el método `unshift()` (descrito en §7.8) para insertar un valor al principio de un array, desplazando los elementos existentes del array a índices superiores. El método `pop()` es lo contrario de `push()`: elimina el último elemento de la matriz y lo devuelve, reduciendo la longitud de una matriz en 1. Del mismo modo, el método `shift()` elimina y devuelve el primer elemento de la matriz, reduciendo la longitud en 1 y desplazando todos los elementos a un índice inferior al actual. Para más información sobre estos métodos, véase §7.8.

Se pueden eliminar elementos del array con el operador `delete`, al igual que se pueden eliminar propiedades de objetos:

```
let a = [1,2,3]; delete a[2]; // a ahora no tiene ningún elemento en el índice 2  
en a // => false: no está definido el índice 2 del array  
a.longitud // => 3: el borrado no afecta a la longitud del array
```

Borrar un elemento del array es similar (pero sutilmente diferente) a asignar `undefined` a ese elemento. Tenga en cuenta que el uso de la eliminación de un elemento de la matriz no altera la propiedad de la longitud y no se desplazan los elementos con índices más altos hacia abajo para llenar el vacío que se deja por la propiedad

eliminada. Si borras un elemento de un array, el array se vuelve escaso.

Como vimos anteriormente, también se pueden eliminar elementos del final de un array simplemente estableciendo la propiedad length a la nueva longitud deseada.

Finalmente, splice() es el método de propósito general para insertar, borrar o reemplazar elementos de la matriz. Altera la propiedad de longitud y desplaza los elementos de la matriz a índices superiores o inferiores según sea necesario. Véase [§7.8](#) para más detalles.

## 7.6 Iteración de matrices

A partir de ES6, la forma más sencilla de recorrer cada uno de los elementos de un array (o cualquier objeto iterable) es con el bucle for/of, que se trató en detalle en [§5.4.4](#):

```
let letters = [... "Hello world"]; // Un array de letras let string = ""; for(let letter of  
letters) { string += letter;  
} string // => "Hola mundo"; reensamblamos el texto original
```

El iterador de matrices incorporado que utiliza el bucle for/of devuelve los elementos de una matriz en orden ascendente. No tiene un comportamiento especial para arrays dispersos y simplemente devuelve undefined para cualquier elemento del array que no exista.

Si quieras utilizar un bucle for/of para un array y necesitas conocer el índice de cada elemento del array, utiliza el método entries() del array, junto con la asignación de desestructuración, así

```
let everyother = ""; for(let [índice, letra] of letras.entradas()) { if (índice % 2 === 0) everyother += letra; // letras en índices pares } everyother // => "Hlowrd"
```

Otra buena forma de iterar matrices es con forEach(). No se trata de una nueva forma del bucle for, sino de un método de array que ofrece un enfoque funcional para la iteración de arrays. Pasas una función al método forEach() de un array, y forEach() invoca tu función una vez en cada elemento del array:

```
let uppercase = ""; letters.forEach(letter => { // Tenga en cuenta la sintaxis de la función de flecha uppercase += letra.toUpperCase(); }); uppercase // => "HELLO WORLD"
```

Como es de esperar, forEach() itera el array en orden, y de hecho pasa el índice del array a su función como segundo argumento, lo que es ocasionalmente útil. A diferencia del bucle for/of, forEach() es consciente de los arrays dispersos y no invoca su función para los elementos que no están allí.

[§7.8.1](#) documenta el método forEach() con más detalle. Esta sección también cubre métodos relacionados como map() y filter() que realizan tipos especializados de iteración de matrices.

También puede recorrer los elementos de un array con un bucle for a la antigua usanza ([§5.4.3](#)):

```
let vowels = ""; for(let i = 0; i < letters.length; i++) { // Por cada índice de la matriz let  
letter = letters[i]; // Obtener el elemento en ese índice if (/[aeiou]/.test(letter)) { //  
Utilizar una expresión regular test vowels += letter; // Si es una vocal, recordarlo }  
} vocales //=> "eoo"
```

En los bucles anidados, o en otros contextos en los que el rendimiento es crítico, a veces se puede ver este bucle básico de iteración de arrays escrito de forma que la longitud del array sólo se busque una vez en lugar de en cada iteración. Ambas formas de bucle for son idiomáticas, aunque no son particularmente comunes, y con los intérpretes modernos de JavaScript, no está nada claro que tengan algún impacto en el rendimiento:

```
// Guarda la longitud del array en una variable local for(let i = 0, len = letras.  
longitud; i < len; i++) { // el cuerpo del bucle sigue siendo el mismo  
}
```

```
// Iterar hacia atrás desde el final de la matriz hasta el principio for(let i = letras.  
longitud-1; i >= 0; i--) { // el cuerpo del bucle sigue siendo el mismo  
}
```

Estos ejemplos suponen que el array es denso y que todos los elementos contienen datos válidos. Si este no es el caso, deberías probar los elementos del array antes de utilizarlos. Si quieres omitir los elementos no definidos o inexistentes, puedes escribir:

```
for(let i = 0; i < a.length; i++) { if (a[i] === undefined) continue; // Skip undefined +  
nonexistent elements // loop body here  
}
```

## 7.7 Matrices multidimensionales

JavaScript no admite verdaderas matrices multidimensionales, pero puede aproximarse a ellas con matrices de matrices. Para acceder a un valor de una matriz de matrices, basta con utilizar dos veces el operador `[]`. Por ejemplo, supongamos que la variable `matrix` es una matriz de matrices de números. Cada elemento de `matrix[x]` es una matriz de números. Para acceder a un número concreto dentro de esta matriz, se escribiría `matrix[x][y]`. He aquí un ejemplo concreto que utiliza una matriz bidimensional como tabla de multiplicación:

```
// Crear un array multidimensional let table = new Array(10); // 10 filas de la tabla
for(let i = 0; i < table.length; i++) { table[i] = new Array(10); // Cada fila tiene 10 columnas }

// Inicializar la matriz for(let row = 0; row < table.length; row++) { for(let col = 0; col < table[row].length; col++) { table[row][col] = row*col; } }

// Utiliza la matriz multidimensional para calcular 5*7 tabla[5][7] // => 35
```

## 7.8 Métodos de la matriz

Las secciones anteriores se han centrado en la sintaxis básica de JavaScript para trabajar con arrays. Sin embargo, en general, los métodos definidos por la clase `Array` son los más potentes. Las siguientes secciones documentan estos métodos. Mientras lees sobre estos métodos, ten en cuenta que algunos de ellos modifican el array al que son llamados y otros dejan el array sin cambios. Algunos métodos devuelven un array: a veces, se trata de

un nuevo array, y el original no se modifica. Otras veces, un método modificará el array en su lugar y también devolverá una referencia al array modificado.

Cada una de las subsecciones que siguen cubre un grupo de métodos de array relacionados:

- Los métodos de iteración hacen un bucle sobre los elementos de una matriz, normalmente invocando una función que se especifica en cada uno de esos elementos.
- Los métodos de pila y cola añaden y eliminan elementos de la matriz al principio y al final de la misma.
- Los métodos de submatriz sirven para extraer, borrar, insertar, llenar y copiar regiones contiguas de una matriz mayor.
- Los métodos de búsqueda y ordenación sirven para localizar elementos dentro de un array y para ordenar los elementos de un array.

Las siguientes subsecciones también cubren los métodos estáticos de la clase `Array` y algunos métodos misceláneos para concatenar arrays y convertir arrays en cadenas.

### 7.8.1 Métodos de Iteración de Matrices

Los métodos descritos en esta sección iteran sobre los arrays pasando los elementos del array, en orden, a una función que usted suministra, y proporcionan formas convenientes de iterar, mapear, filtrar, probar y reducir arrays.

Sin embargo, antes de explicar los métodos en detalle, conviene hacer algunas generalizaciones sobre ellos. En primer lugar, todos

estos métodos aceptan una función como primer argumento e invocan esa función una vez por cada elemento (o algunos elementos) del array. Si la matriz es dispersa, la función que se pasa no se invoca para los elementos inexistentes. En la mayoría de los casos, la función que se suministra se invoca con tres argumentos: el valor del elemento del array, el índice del elemento del array y el propio array. A menudo, sólo se necesita el primero de estos argumentos y se pueden ignorar el segundo y el tercero.

La mayoría de los métodos de iteración descritos en las siguientes subsecciones aceptan un segundo argumento opcional. Si se especifica, la función se invoca como si fuera un método de este segundo argumento. Es decir, el segundo argumento que se pasa se convierte en el valor de la palabra clave `this` dentro de la función que se pasa como primer argumento. El valor de retorno de la función que se pasa suele ser importante, pero diferentes métodos manejan el valor de retorno de diferentes maneras.

Ninguno de los métodos descritos aquí modifica el array sobre el que se invoca (aunque la función que se pasa puede modificar el array, por supuesto).

Cada una de estas funciones se invoca con una función como primer argumento, y es muy común definir esa función en línea como parte de la expresión de invocación del método en lugar de utilizar una función existente que esté definida en otra parte. La sintaxis de la función flecha ([véase §8.1.3](#)) funciona especialmente bien con estos métodos, y la utilizaremos en los ejemplos que siguen.

## FOREACH()

El método forEach() itera a través de una matriz, invocando una función que se especifica para cada elemento. Como hemos descrito, pasas la función como primer argumento a forEach(). forEach() entonces invoca tu función con tres argumentos: el valor del elemento del array, el índice del elemento del array, y el propio array. Si sólo le interesa el valor del elemento de la matriz, puede escribir una función con un solo parámetro - los argumentos adicionales serán ignorados:

```
let data = [1,2,3,4,5], sum = 0; // Calcula la suma de los elementos del array data.  
forEach(value => { sum += value; }); // sum == 15  
  
// Ahora incrementa cada elemento del array  
data.forEach(function(v, i, a) { a[i] = v + 1; }); // data == [2,3,4,5,6]
```

Tenga en cuenta que forEach() no proporciona una forma de terminar la iteración antes de que todos los elementos hayan sido pasados a la función. Es decir, no hay un equivalente a la sentencia break que se puede utilizar con un bucle for normal.

## MAPA()

El método map() pasa cada elemento del array sobre el que se invoca a la función que se especifica y devuelve un array que contiene los valores devueltos por su función. Por ejemplo:

```
dejemos a = [1, 2, 3];  
a. map(x => x*x) // => [1, 4, 9]: la función toma la entrada x y devuelve x*x
```

La función que se pasa a map() se invoca de la misma manera que una función pasada a forEach(). Sin embargo, para el método map(), la función que se pasa debe devolver un valor. Tenga en

cuenta que map() devuelve un nuevo array: no modifica el array sobre el que se invoca. Si ese array es escaso, su función no será llamada para los elementos que faltan, pero el array devuelto será escaso de la misma manera que el array original: tendrá la misma longitud y los mismos elementos que faltan.

## FILTRO()

El método filter() devuelve un array que contiene un subconjunto de los elementos del array sobre el que se invoca. La función que se le pasa debe ser un predicado: una función que devuelve verdadero o falso. El predicado se invoca igual que para forEach() y map(). Si el valor de retorno es verdadero, o un valor que se convierte en verdadero, entonces el elemento pasado al predicado es un miembro del subconjunto y se añade al array que se convertirá en el valor de retorno. Ejemplos:

```
dejemos a = [5, 4, 3, 2, 1];
a. filter(x => x < 3) // => [2, 1]; valores inferiores a 3
a. filter((x,i) => i%2 === 0) // => [5, 3, 1]; cualquier otro valor
```

Tenga en cuenta que filter() omite los elementos que faltan en las matrices dispersas y que su valor de retorno es siempre denso. Para cerrar los huecos en una matriz dispersa, puede hacer lo siguiente:

```
let dense = sparse. filter(() => true);
```

Y para cerrar los huecos y eliminar los elementos indefinidos y nulos, se puede utilizar el filtro, así:

```
a = a. filter(x => x !== undefined && x !== null);
```

## FIND() Y FINDINDEX()

Los métodos `find()` y `findIndex()` son como `filter()` en el sentido de que iteran a través de su matriz buscando elementos para los que su función de predicado devuelve un valor verdadero. Sin embargo, a diferencia de `filter()`, estos dos métodos dejan de iterar la primera vez que el predicado encuentra un elemento. Cuando esto ocurre, `find()` devuelve el elemento coincidente, y `findIndex()` devuelve el índice del elemento coincidente. Si no se encuentra ningún elemento coincidente, `find()` devuelve `undefined` y `findIndex()` devuelve `-1`:

```
dejemos a = [1,2,3,4,5];
a. findIndex(x => x === 3) // => 2; el valor 3 aparece en el índice 2
a. findIndex(x => x < 0) // => -1; no hay números negativos en la matriz
a. find(x => x % 5 === 0) // => 5: es un múltiplo de 5
a. find(x => x % 7 === 0) // => undefined: no hay múltiplos de 7 en el array
```

## EVERY() Y SOME()

Los métodos `every()` y `some()` son predicados de matrices: aplican una función de predicado que se especifica a los elementos de la matriz, y luego devuelven `true` o `false`.

El método `every()` es como el cuantificador matemático "para todos"  $\forall$ : devuelve verdadero si y sólo si su función de predicado devuelve verdadero para todos los elementos de la matriz:

```
dejemos a = [1,2,3,4,5];
a. every(x => x < 10) // => true: todos los valores son < 10.
a. every(x => x % 2 === 0) // => false: no todos los valores son pares.
```

El método `some()` es como el cuantificador matemático "existe"  $\exists$ : devuelve verdadero si existe al menos un elemento en el array

para el que el predicado devuelve verdadero y devuelve falso si y sólo si el predicado devuelve falso para todos los elementos del array:

```
dejemos a = [1,2,3,4,5];
a. some(x => x%2==0) // => true; a tiene algunos números pares.
a. some(isNaN) // => false; a no tiene números.
```

Tenga en cuenta que tanto every() como some() dejan de iterar los elementos del array en cuanto saben qué valor devolver. some() devuelve true la primera vez que su predicado devuelve <code>true</code> y sólo itera por todo el array si su predicado siempre devuelve false. every() es lo contrario: devuelve false la primera vez que su predicado devuelve false y sólo itera todos los elementos si su predicado siempre devuelve true. Observe también que, por convención matemática, every() devuelve true y some devuelve false cuando se invoca sobre un array vacío.

## REDUCE() Y REDUCERIGHT()

Los métodos reduce() y reduceRight() combinan los métodos elementos de un array, utilizando la función que se especifique, para producir un único valor. Esta es una operación común en la programación funcional y también recibe los nombres de "inyectar" y "plegar". Los ejemplos ayudan a ilustrar cómo funciona:

```
dejemos a = [1,2,3,4,5];
a. reduce((x,y) => x+y, 0) // => 15; la suma de los valores
a. reduce((x,y) => x*y, 1) // => 120; el producto de los valores
a. reduce((x,y) => (x > y) ? x : y) // => 5; el mayor de los valores
```

reduce() toma dos argumentos. El primero es la función que realiza la operación de reducción. La tarea de esta función de reducción es combinar o reducir de alguna manera dos valores en un solo valor y devolver ese valor reducido. En los ejemplos que hemos mostrado aquí, las funciones combinan dos valores sumándolos, multiplicándolos y eligiendo el mayor. El segundo argumento (opcional) es un valor inicial que se pasa a la función.

Las funciones utilizadas con reduce() son diferentes a las utilizadas con forEach() y map(). El valor familiar, el índice y los valores del array se pasan como segundo, tercer y cuarto argumento. El primer argumento es el resultado acumulado de la reducción hasta el momento. En la primera llamada a la función, este primer argumento es el valor inicial que se pasó como segundo argumento a reduce(). En las llamadas posteriores, es el valor devuelto por la invocación anterior de la función. En el primer ejemplo, la función de reducción se llama primero con los argumentos 0 y 1. Los suma y devuelve 1. A continuación, se llama de nuevo con los argumentos 1 y 2 y devuelve 3. A continuación, calcula  $3+3=6$ , luego  $6+4=10$ , y finalmente  $10+5=15$ . Este valor final, 15, se convierte en el valor de retorno de reduce().

Puede que haya notado que la tercera llamada a reduce() en este ejemplo sólo tiene un único argumento: no hay ningún valor inicial especificado. Cuando se invoca a reduce() de esta manera sin valor inicial, se utiliza el primer elemento del array como valor inicial. Esto significa que la primera llamada a la función de reducción tendrá el primer y segundo elemento del array como primer y

segundo argumento. En los ejemplos de suma y producto, podríamos haber omitido el argumento del valor inicial.

Llamar a `reduce()` en una matriz vacía sin argumento de valor inicial provoca un `TypeError`. Si se llama con un solo valor, ya sea una matriz con un elemento y sin valor inicial o una matriz vacía y un valor inicial, simplemente devuelve ese valor sin llamar nunca a la función de reducción.

`reduceRight()` funciona igual que `reduce()`, excepto que procesa la matriz desde el índice más alto al más bajo (de derecha a izquierda), en lugar de hacerlo de menor a mayor. Es posible que desee hacer esto si la operación de reducción tiene asociatividad de derecha a izquierda, por ejemplo:

```
// Calcula 2^(3^4). La exponenciación tiene precedencia de derecha a izquierda
deja a = [2, 3, 4];
a. reduceRight((acc, val) => Math. pow(val, acc)) // =>
2.4178516392292583e+24
```

Tenga en cuenta que ni `reduce()` ni `reduceRight()` aceptan un argumento opcional que especifique el valor de este sobre el que se va a invocar la función de reducción. El argumento opcional valor inicial ocupa su lugar. Consulte el método `Function.bind()` ([§8.7.5](#)) si necesita que su función de reducción sea invocada como un método de un objeto concreto.

Los ejemplos mostrados hasta ahora han sido numéricos para simplificar, pero `reduce()` y `reduceRight()` no están pensados únicamente para cálculos matemáticos. Cualquier función que pueda combinar dos valores (como dos objetos) en un valor del mismo tipo puede utilizarse como función de reducción. Por otra

parte, los algoritmos expresados mediante reducciones de arrays pueden volverse rápidamente complejos y difíciles de entender, y puede que le resulte más fácil leer, escribir y razonar sobre su código si utiliza construcciones de bucle regulares para procesar sus arrays.

## 7.8.2 Aplanar matrices con flat() y flatMap()

En ES2019, el método flat() crea y devuelve una nueva matriz que contiene los mismos elementos que la matriz a la que se llama, excepto que cualquier elemento que sea en sí mismo una matriz se "aplana" en la matriz devuelta. Por ejemplo:

```
[1, [2, 3]]. flat() //=> [1, 2, 3]
[1, [2, [3]]]. flat() //=> [1, 2, [3]]
```

Cuando se llama sin argumentos, flat() aplana un nivel de anidamiento. Los elementos de la matriz original que son a su vez matrices se apilan, pero los elementos de la matriz de *esas* matrices no se apilan. Si desea aplinar más niveles, pase un número a flat():

```
dejemos a = [1, [2, [3, [4]]]];
a. flat(1) //=> [1, 2, [3, [4]]]
a. flat(2) //=> [1, 2, 3, [4]]
a. flat(3) //=> [1, 2, 3, 4]
a. flat(4) //=> [1, 2, 3, 4]
```

El método flatMap() funciona igual que el método map() (véase "[map\(\)](#)"), excepto que la matriz devuelta se aplana automáticamente como si se pasara a flat(). Es decir, llamar a.flatMap(f) es lo mismo (pero más eficiente) que a.map(f).flat():

```
let phrases = ["hola mundo", "la guía definitiva"]; let words = phrases.  
flatMap(phrase => phrase.split(" ")); words // => ["hola", "mundo", "la",  
"definitiva", "guía"];
```

Puede pensar en flatMap() como una generalización de map() que permite que cada elemento de la matriz de entrada se asigne a cualquier número de elementos de la matriz de salida. En particular, flatMap() permite asignar elementos de entrada a una matriz vacía, que se aplana a nada en la matriz de salida:

```
// Asignar números no negativos a sus raíces cuadradas  
[-2, -1, 1, 2].flatMap(x => x < 0 ? [] : Math.sqrt(x)) // =>  
[1, 2**0.5]
```

### 7.8.3 Añadir matrices con concat()

El método concat() crea y devuelve un nuevo array que contiene los elementos del array original sobre el que se ha invocado concat(), seguido de cada uno de los argumentos de concat(). Si alguno de estos argumentos es a su vez un array, entonces son los elementos del array los que se concatenan, no el propio array.

Tenga en cuenta, sin embargo, que concat() no aplana recursivamente matrices de matrices. concat() no modifica la matriz sobre la que se invoca:

```
dejemos a = [1,2,3];  
a.concat(4, 5) // => [1,2,3,4,5]  
a.concat([4,5],[6,7]) // => [1,2,3,4,5,6,7]; las matrices se aplanan  
a.concat(4, [5,[6,7]]) // => [1,2,3,4,5,[6,7]]; pero no las matrices anidadas a // =>  
[1,2,3]; la matriz original no se modifica
```

Tenga en cuenta que concat() hace una nueva copia del array al que se llama. En muchos casos, esto es lo correcto, pero es una operación cara. Si se encuentra escribiendo código como a =

a.concat(x), entonces debería pensar en modificar su matriz en su lugar con push() o splice() en lugar de crear una nueva.

#### 7.8.4 Pilas y colas con push(), pop(), shift() y unshift()

Los métodos push() y pop() permiten trabajar con arrays como si fueran pilas. El método push() añade uno o más elementos nuevos al final de una matriz y devuelve la nueva longitud de la misma. A diferencia de concat(), push() no aplana los argumentos de la matriz. El método pop() hace lo contrario: elimina el último elemento de una matriz, disminuye la longitud de la matriz y devuelve el valor que ha eliminado. Tenga en cuenta que ambos métodos modifican la matriz en su lugar. La combinación de push() y pop() permite utilizar una matriz de JavaScript para implementar una pila de primeras entradas y últimas salidas. Por ejemplo:

```
let stack = []; // stack == []
stack.push(1,2); // stack == [1,2];
stack.pop(); // stack == [1];
stack.push(3); // stack == [1,3];
stack.pop(); // pila == [1];
stack.push([4,5]); // pila == [1,[4,5]];
stack.pop(); // pila == [1];
stack.pop(); // pila == [];
stack.pop(); // pila == 1
```

El método push() no aplana un array que le pases, pero si quieres empujar todos los elementos de un array a otro array, puedes usar el operador spread ([§8.3.4](#)) para aplanarlo explícitamente:

a. `push(... valores);`

Los métodos unshift() y shift() se comportan de forma muy parecida a push() y pop(), excepto que insertan y eliminan elementos desde el principio de una matriz en lugar de hacerlo desde el final. unshift() añade un elemento o elementos al principio de la matriz, desplaza los elementos existentes de la matriz a índices más altos para hacer espacio, y devuelve la nueva longitud de la matriz. shift() elimina y devuelve el primer elemento

de la matriz, desplazando todos los elementos subsiguientes un lugar hacia abajo para ocupar el nuevo espacio vacante al principio de la matriz. Podrías usar `unshift()` y `shift()` para implementar una pila, pero sería menos eficiente que usar `push()` y `pop()` porque los elementos del array necesitan ser desplazados hacia arriba o hacia abajo cada vez que se añade o se elimina un elemento al principio del array. En su lugar, sin embargo, puedes implementar una estructura de datos de cola utilizando `push()` para añadir elementos al final de un array y `shift()` para quitarlos del principio del array:

```
let q = [] // q == []
q.push(1,2) // q == [1,2]
q.shift() // q == [2]; devuelve 1
q.push(3) // q == [2, 3]
q.shift() // q == [3]; devuelve 2
q.shift() // q == []; devuelve 3
```

Hay una característica de `unshift()` que merece la pena destacar porque puede resultarle sorprendente. Cuando se pasan varios argumentos a `unshift()`, se insertan todos a la vez, lo que significa que acaban en la matriz en un orden diferente al que tendrían si se insertaran de uno en uno:

```
let a = [] // a == []
a.unshift(1) // a == [1]
a.unshift(2) // a == [2, 1] a = [] // a == []
a.unshift(1,2)// a == [1, 2]
```

### 7.8.5 Submatriz con `slice()`, `splice()`, `fill()` y `copyWithin()`

Las matrices definen una serie de métodos que funcionan en regiones contiguas, o submatrices o "rebanadas" de una matriz.

Las siguientes secciones describen métodos para extraer, reemplazar, llenar y copiar rebanadas.

## SLICE()

El método slice() devuelve una *porción*, o submatriz, de la matriz especificada. Sus dos argumentos especifican el inicio y el final de la porción a devolver. La matriz devuelta contiene el elemento especificado por el primer argumento y todos los elementos subsiguientes hasta, pero sin incluir, el elemento especificado por el segundo argumento. Si sólo se especifica un argumento, la matriz devuelta contiene todos los elementos desde la posición inicial hasta el final de la matriz. Si cualquiera de los argumentos es negativo, especifica un elemento del array relativo a la longitud del mismo. Un argumento de -1, por ejemplo, especifica el último elemento de la matriz, y un argumento de -2 especifica el elemento anterior a ese. Tenga en cuenta que slice() no modifica la matriz sobre la que se invoca. He aquí algunos ejemplos:

```
dejemos a = [1,2,3,4,5];
a. slice(0,3); // Devuelve [1,2,3]
a. slice(3); // Devuelve [4,5]
a. slice(1,-1); // Devuelve [2,3,4]
a. slice(-3,-2); // Devuelve [3]
```

## SPLICE()

splice() es un método de propósito general para insertar o eliminar elementos de una matriz. A diferencia de slice() y concat(), splice() modifica la matriz sobre la que se invoca. Tenga en cuenta que splice() y slice() tienen nombres muy similares pero realizan operaciones sustancialmente diferentes.

splice() puede borrar elementos de una matriz, insertar nuevos elementos en una matriz o realizar ambas operaciones al mismo tiempo. Los elementos de la matriz que vienen después del punto de inserción o borrado tienen sus índices aumentados o disminuidos según sea necesario para que permanezcan contiguos con el resto de la matriz. El primer argumento de splice() especifica la posición de la matriz en la que debe comenzar la inserción y/o eliminación. El segundo argumento especifica el número de elementos que deben ser eliminados (empalmados) de la matriz. ( Nótese que esta es otra diferencia entre estos dos métodos. El segundo argumento de slice() es una posición final. El segundo argumento de splice() es una longitud). Si se omite este segundo argumento, se eliminan todos los elementos del array desde el elemento inicial hasta el final del array. splice() devuelve un array con los elementos eliminados, o un array vacío si no se ha eliminado ningún elemento. Por ejemplo:

```
dejemos a = [1,2,3,4,5,6,7,8];
a. splice(4) //=> [5,6,7,8]; a es ahora [1,2,3,4]
a. splice(1,2) //=> [2,3]; a es ahora [1,4]
a. splice(1,1) //=> [4]; a es ahora [1]
```

Los dos primeros argumentos de splice() especifican los elementos de la matriz que se van a eliminar. Estos argumentos pueden ir seguidos de cualquier número de argumentos adicionales que especifiquen los elementos que se van a insertar en la matriz, empezando por la posición especificada por el primer argumento. Por ejemplo:

```
dejemos a = [1,2,3,4,5];
a. splice(2,0, "a", "b") //=> []; a es ahora [1,2, "a", "b",3,4,5]
a. splice(2,2,[1,2],3) //=> ["a", "b"]; a es ahora [1,2, [1,2],3,3,4,5]
```

Tenga en cuenta que, a diferencia de concat(), splice() inserta las propias matrices, no los elementos de esas matrices.

## FILL()

El método fill() establece los elementos de una matriz, o una porción de una matriz, a un valor especificado. Mutá la matriz sobre la que se llama, y también devuelve la matriz modificada:

```
let a = new Array(5); // Comienza sin elementos y con una longitud de 5
a. fill(0) // => [0,0,0,0,0]; rellena el array con ceros
a. fill(9, 1) // => [0,9,9,9,9]; llenar con 9 empezando por el índice 1
a. fill(8, 2, -1) // => [0,9,8,8,9]; llenar con 8 en los índices 2, 3
```

El primer argumento de fill() es el valor al que se van a asignar los elementos del array. El segundo argumento opcional especifica el índice inicial. Si se omite, el llenado comienza en el índice 0. El tercer argumento opcional especifica el índice final: se llenarán los elementos de la matriz hasta este índice, pero sin incluirlo. Si se omite este argumento, la matriz se llena desde el índice inicial hasta el final. Se pueden especificar índices relativos al final de la matriz pasando números negativos, al igual que para slice().

## COPYWITHIN()

copyWithin() copia una porción de una matriz a una nueva posición dentro de la misma. Modifica la matriz en su lugar y devuelve la matriz modificada, pero no cambiará la longitud de la matriz. El primer argumento especifica el índice de destino al que se copiará el primer elemento. El segundo argumento especifica el índice del primer elemento a copiar. Si se omite este segundo argumento, se utiliza 0. El tercer argumento especifica el final de la porción de elementos a copiar. Si se omite, se utiliza la longitud del

array. Se copiarán los elementos desde el índice inicial hasta el índice final, pero sin incluirlo. Se pueden especificar índices relativos al final de la matriz pasando números negativos, al igual que para slice():

```
dejemos a = [1,2,3,4,5];
a. copyWithin(1) //=> [1,1,2,3,4]: copia los elementos del array uno arriba
a. copyWithin(2, 3, 5) //=> [1,1,3,4,4]: copia los 2 últimos elementos al índice 2
a. copyWithin(0, -2) //=> [4,4,3,4,4]: los desplazamientos negativos también
funcionan
```

copyWithin() está pensado como un método de alto rendimiento que es particularmente útil con arrays tipados (véase §11.2). Se basa en la función memmove() de la biblioteca estándar de C. Tenga en cuenta que la copia funcionará correctamente incluso si hay solapamiento entre las regiones de origen y destino.

### 7.8.6 Métodos de búsqueda y ordenación de matrices

Las matrices implementan los métodos indexOf(), lastIndexOf() e includes() que son similares a los métodos del mismo nombre de las cadenas. También existen los métodos sort() y reverse() para reordenar los elementos de un array. Estos métodos se describen en las subsecciones siguientes.

#### INDEXOF() Y LASTINDEXOF()

indexOf() y lastIndexOf() buscan en una matriz un elemento con un valor especificado y devuelven el índice del primer elemento encontrado, o -1 si no se encuentra ninguno. indexOf() busca en la matriz desde el principio hasta el final, y lastIndexOf() busca desde el final hasta el principio:

```
dejemos a = [0,1,2,1,0];
```

```
a. indexOf(1) //=> 1: a[1] es 1  
a. lastIndexOf(1) //=> 3: a[3] es 1  
a. indexOf(3) //=> -1: ningún elemento tiene valor 3
```

indexOf() y lastIndexOf() comparan su argumento con el utilizando el equivalente al operador ===. Si su matriz contiene objetos en lugar de valores primitivos, estos métodos comprueban si dos referencias se refieren exactamente al mismo objeto. Si quieres ver realmente el contenido de un objeto, intenta utilizar el método find() con tu propia función de predicado personalizada.

indexOf() y lastIndexOf() toman un segundo opcional que especifica el índice de la matriz en el que se inicia la búsqueda. Si se omite este argumento, indexOf() empieza por el principio y lastIndexOf() empieza por el final. Se permiten valores negativos para el segundo argumento y se tratan como un desplazamiento desde el final de la matriz, al igual que para el método slice(): un valor de -1, por ejemplo, especifica el último elemento de la matriz.

La siguiente función busca en un array un valor especificado y devuelve un array con *todos los* índices que coinciden. Esto demuestra cómo el segundo argumento de indexOf() puede utilizarse para encontrar coincidencias más allá del primero.

```
// Encuentra todas las apariciones de un valor x en un array a y devuelve un array //  
de índices coincidentes function findall(a, x) { let results = [], // El array de índices  
que devolveremos len = a.length, // La longitud del array a buscar pos = 0; // La  
posición a buscar desde while(pos < len) { // Mientras más elementos a buscar...  
pos = a.indexOf(x, pos); // Buscar si (pos === -1) break; // Si no se encuentra  
nada, hemos terminado. results.push(pos); // Si no, guardar el índice en el array  
pos = pos + 1; // Y empezar la siguiente búsqueda en el siguiente elemento } return  
results; // Devolver el array de índices }
```

Tenga en cuenta que las cadenas tienen los métodos `indexOf()` y `lastIndexOf()` que funcionan como estos métodos de matriz, excepto que un segundo argumento negativo se trata como cero.

## INCLUYE()

El método ES2016 `includes()` toma un único argumento y devuelve `true` si el array contiene ese valor o `false` en caso contrario. No indica el índice del valor, sólo si existe. El método `includes()` es efectivamente una prueba de pertenencia a un conjunto para los arrays. Tenga en cuenta, sin embargo, que los arrays no son una representación eficiente para los conjuntos, y si está trabajando con más de unos pocos elementos, debe utilizar un objeto Set real ([§11.1.1](#)).

El método `includes()` es ligeramente diferente al método `indexOf()` en un aspecto importante. `indexOf()` comprueba la igualdad utilizando el mismo algoritmo que el operador `==`, y ese algoritmo de igualdad considera que el valor no numérico es diferente de cualquier otro valor, incluido él mismo. `includes()` utiliza una versión ligeramente diferente de la igualdad que sí considera que `NaN` es igual a sí mismo. Esto significa que `indexOf()` no detectará el valor `NaN` en una matriz, pero `includes()` sí:

```
deje a = [1,true,3,NaN];
a. includes(true) // => true
a. includes(2) // => false
a. includes(NaN) // => true
a. indexOf(NaN) // => -1; indexOf no puede encontrar NaN
```

## SORT()

sort() ordena los elementos de una matriz en su lugar y devuelve la matriz ordenada. Cuando se llama a sort() sin argumentos, ordena los elementos de la matriz en orden alfabético (convirtiéndolos temporalmente en cadenas para realizar la comparación, si es necesario):

```
let a = ["plátano", "cereza", "manzana"];
a.sort(); // a == ["manzana", "plátano", "cereza"]
```

Si un array contiene elementos no definidos, se ordenan al final del array.

Para ordenar una matriz en un orden distinto al alfabético, debe pasar una función de comparación como argumento a sort(). Esta función decide cuál de sus dos argumentos debe aparecer primero en la matriz ordenada. Si el primer argumento debe aparecer antes que el segundo, la función de comparación debe devolver un número menor que cero. Si el primer argumento debe aparecer después del segundo en la matriz ordenada, la función debe devolver un número mayor que cero. Y si los dos valores son equivalentes (es decir, si su orden es irrelevante), la función de comparación debe devolver 0. Así, por ejemplo, para ordenar los elementos de la matriz en orden numérico en lugar de alfabético, podría hacer lo siguiente:

```
dejemos a = [33, 4, 1111, 222];
a.sort(); // a == [1111, 222, 33, 4]; orden alfabético
a.sort(function(a,b) { // Pasa una función comparadora return a-b; // Devuelve < 0, 0, o > 0, dependiendo del orden });
a.sort((a,b) => b-a); // a == [1111, 222, 33, 4]; invertir el orden numérico
```

Como otro ejemplo de ordenación de elementos de matrices, puede realizar una ordenación alfabética sensible a mayúsculas y minúsculas en una matriz de cadenas pasando una función de comparación que convierta ambos argumentos a minúsculas (con el método `toLowerCase()`) antes de compararlos:

```
let a = ["hormiga", "bicho", "gato", "perro"];
a.sort(); // a == ["Bicho", "Perro", "hormiga", "gato"]; ordenación sensible a
mayúsculas y minúsculas
a.sort(function(s,t) { let a = s.
toLowerCase(); let b = t.toLowerCase();
if (a < b) return -1; if (a > b) return 1;
return 0;
}); // a == ["hormiga", "bicho", "gato", "perro"]; ordenación sin distinción de
mayúsculas y minúsculas
```

## REVERSE()

El método `reverse()` invierte el orden de los elementos de un array y devuelve el array invertido. Lo hace en su lugar; en otras palabras, no crea un nuevo array con los elementos reordenados sino que los reordena en el array ya existente:

```
dejemos a = [1,2,3];
a.reverse(); // a == [3,2,1]
```

## 7.8.7 Conversiones de matrices a cadenas

La clase `Array` define tres métodos que pueden convertir arrays en cadenas, lo que generalmente es algo que podrías hacer al crear mensajes de registro y de error. (Si quieres guardar el contenido de un array en forma textual para reutilizarlo posteriormente, serializa el array con `JSON.stringify()` [§6.8] en lugar de utilizar los métodos descritos aquí).

El método `join()` convierte todos los elementos de una matriz en cadenas y las concatena, devolviendo la cadena resultante. Puede especificar una cadena opcional que separe los elementos en la cadena resultante. Si no se especifica ninguna cadena separadora, se utiliza una coma:

```
dejemos a = [1, 2, 3];
a.join() //=> "1,2,3"
a.join(" ") //=> "1 2 3"
a.join("") //=> "123" let b = new Array(10); // Un array de longitud 10 sin elementos
b.join("-") //=> "-----": una cadena de 9 guiones
```

El método `join()` es el inverso del método `String.split()`, que crea una matriz al dividir una cadena en trozos.

Las matrices, como todos los objetos de JavaScript, tienen un método `toString()`. Para un array, este método funciona igual que el método `join()` sin argumentos:

```
[1,2,3].toString() //=> "1,2,3"
["a", "b", "c"].toString() //=> "a,b,c"
[1, [2, "c"]].toString() //=> "1,[2,c]"
```

Tenga en cuenta que la salida no incluye corchetes ni ningún otro tipo de delimitador alrededor del valor de la matriz.

`toLocaleString()` es la versión localizada de `toString()`. Convierte cada elemento de la matriz en una cadena llamando al método `toLocaleString()` del elemento, y luego concatena las cadenas resultantes utilizando una cadena separadora específica de la localidad (y definida por la implementación).

## 7.8.8 Funciones estáticas de las matrices

Además de los métodos de arrays que ya hemos documentado, la clase Array también define tres funciones estáticas que puedes invocar a través del constructor de Array en lugar de en los arrays. Array.of() y Array.from() son métodos de fábrica para crear nuevos arrays. Fueron documentados en §7.1.4 y §7.1.5.

La otra función estática de array es Array.isArray(), que es útil para determinar si un valor desconocido es un array o no:

```
Array.isArray([]) //=> true  
Array.isArray({}) //=> false
```

## 7.9 Objetos similares a matrices

Como hemos visto, las matrices de JavaScript tienen algunas características especiales que no tienen otros objetos:

- La propiedad de longitud se actualiza automáticamente a medida que se añaden nuevos elementos a la lista.
- Al establecer la longitud en un valor menor se trunca la matriz.
- Los arrays heredan métodos útiles de Array.prototype.

Array.isArray() devuelve true para los arrays.

Estas son las características que hacen que los arrays de JavaScript se diferencien de los objetos normales. Pero no son las características esenciales que definen un array. A menudo es perfectamente razonable tratar cualquier objeto con una propiedad de longitud numérica y las correspondientes propiedades de enteros no negativos como un tipo de array.

Estos objetos "tipo array" aparecen ocasionalmente en la práctica, y aunque no puedes invocar directamente métodos de array sobre ellos o esperar un comportamiento especial de la propiedad `length`, puedes iterar a través de ellos con el mismo código que usarías para un array real. Resulta que muchos algoritmos de arrays funcionan tan bien con objetos tipo array como con arrays reales. Esto es especialmente cierto si tus algoritmos tratan el array como de sólo lectura o si al menos dejan la longitud del array sin cambios.

El siguiente código toma un objeto normal, añade propiedades para convertirlo en un objeto similar a un array, y luego itera a través de los "elementos" del pseudoarray resultante:

```
let a = {}; // Comienza con un objeto regular vacío

// Añade propiedades para hacerlo "tipo array" let i = 0;
while(i < 10) { a[i] = i * i; i++; }

a.longitud = i;

// Ahora itera a través de ella como si fuera una matriz real let total = 0;
for(let j = 0; j < a.length; j++) { total += a[j];
}
```

En el lado del cliente de JavaScript, hay una serie de métodos para trabajar con

documentos HTML (como `document.querySelectorAll()`, por ejemplo) devuelven objetos tipo array. Esta es una función que podría utilizar para comprobar si los objetos funcionan como arrays:

```
// Determina si o es un objeto tipo array.  
// Las cadenas y las funciones tienen propiedades de longitud numérica, pero son  
// excluidos por la prueba typeof. En el lado del cliente de JavaScript,  
// los nodos tienen una propiedad de longitud numérica, y pueden necesitar ser  
// excluidos // con una prueba adicional o.nodeType !== 3.  
function isArrayLike(o) { if (o && // o no es null, undefined, etc.  
    typeof o === "object" && // o es un objeto Number. isFinite(o.length) && //  
    o.length es un número finito  
    o.length >= 0 && // o.length es no negativo Number. isInteger(o.length) && //  
    o.length es un entero  
    o.length < 4294967295) { // o.length < 2^32 - 1 return true; // Entonces o es  
    // arraylike. } else { return false; // En caso contrario no lo es. }  
}
```

Veremos en una sección posterior que las cadenas se comportan como arrays.

Sin embargo, las pruebas como ésta para objetos tipo array suelen devolver false para las cadenas; normalmente es mejor manejarlas como cadenas, no como arrays.

La mayoría de los métodos de arrays de JavaScript se definen a propósito para que sean genéricos y funcionen correctamente cuando se apliquen a objetos tipo array, además de a arrays reales.

Dado que los objetos tipo array no heredan de `Array.prototype`, no se pueden invocar métodos de array directamente. Sin embargo, puedes invocarlos indirectamente usando el método `Function.call` (ver §8.7.4 para más detalles):

```
let a = {"0": "a", "1": "b", "2": "c", length: 3}; // Un objeto tipo array Array.  
prototype.join.call(a, "+") //=>  
"a+b+c"  
Array.prototype.map.call(a, x => x.toUpperCase()) //=>  
["A", "B", "C"] Array.prototype.slice.call(a, 0) //=> ["a", "b", "c"]: true array copy  
Array.from(a) // => ["a", "b", "c"]:  
facilitar la copia de matrices
```

La penúltima línea de este código invoca el método `Array slice()` en un objeto tipo array para copiar los elementos de ese objeto en un verdadero objeto array. Este es un truco idiomático que existe en mucho código heredado, pero que ahora es mucho más fácil de hacer con `Array.from()`.

## 7.10 Cadenas como matrices

Las cadenas de JavaScript se comportan como matrices de sólo lectura de caracteres Unicode UTF-16. En lugar de acceder a caracteres individuales con el método `charAt()`, puede utilizar corchetes:

```
deje s = "test";  
s.charAt(0) //=> "t" s[1] //=> "e"
```

El operador `typeof` sigue devolviendo "string" para las cadenas, por supuesto, y el método `Array.isArray()` devuelve `false` si se le pasa una cadena.

La principal ventaja de las cadenas indexables es simplemente que podemos sustituir las llamadas a `charAt()` por corchetes, que son más concisos y legibles, y potencialmente más eficientes. Sin embargo, el hecho de que las cadenas se comporten como arrays

también significa que podemos aplicarles métodos genéricos de arrays. Por ejemplo:

```
Array.prototype.join("JavaScript", " ") //=> "Java  
Script"
```

Tenga en cuenta que las cadenas son valores inmutables, por lo que cuando se tratan como arrays, son arrays de sólo lectura. Los métodos de matrices como push(), sort(), reverse() y splice() modifican una matriz en su lugar y no funcionan con cadenas. Sin embargo, el intento de modificar una cadena mediante un método de matriz no provoca un error: simplemente falla en silencio.

## 7.11 Resumen

Este capítulo ha cubierto los arrays de JavaScript en profundidad, incluyendo detalles esotéricos sobre arrays dispersos y objetos tipo array. Los principales puntos a tomar de este capítulo son:

- Los literales de las matrices se escriben como listas de valores separadas por comas entre corchetes.
- El acceso a los elementos individuales de la matriz se realiza especificando el índice de la matriz deseado entre corchetes.
- El bucle for/of y el operador de propagación ... introducidos en ES6 son formas especialmente útiles de iterar matrices.
- La clase Array define un rico conjunto de métodos para manipular arrays, y deberías asegurarte de familiarizarte con la API Array.



# Capítulo 8. Funciones

---

Este capítulo cubre las funciones de JavaScript. Las funciones son un bloque de construcción fundamental para los programas de JavaScript y una característica común en casi todos los lenguajes de programación. Es posible que ya estés familiarizado con el concepto de función bajo un nombre como *subrutina* o *procedimiento*.

Una *función* es un bloque de código JavaScript que se define una vez pero que puede ejecutarse, o *invocarse*, cualquier número de veces. Las funciones de JavaScript están *parametrizadas*: una definición de función puede incluir una lista de identificadores, conocidos como *parámetros*, que funcionan como variables locales para el cuerpo de la función. Las invocaciones de funciones proporcionan valores, o *argumentos*, para los parámetros de la función. Las funciones suelen utilizar los valores de sus argumentos para calcular un *valor de retorno* que se convierte en el valor de la expresión de la invocación de la función. Además de los argumentos, cada invocación tiene otro valor -el *contexto de la invocación*- que es el valor de la palabra clave `this`.

Si una función se asigna a una propiedad de un objeto, se conoce como un *método* de ese objeto. Cuando se invoca una función *sobre* o *a través de un objeto*, ese objeto es el contexto de invocación o este valor para la función. Las funciones diseñadas

para inicializar un objeto recién creado se llaman *constructores*. Los constructores se describieron en §6.2 y se volverán a tratar en el capítulo 9.

En JavaScript, las funciones son objetos, y pueden ser manipuladas por los programas. JavaScript puede asignar funciones a variables y pasárlas a otras funciones, por ejemplo. Como las funciones son objetos, se pueden establecer propiedades en ellas e incluso invocar métodos en ellas.

Las definiciones de funciones de JavaScript pueden anidarse dentro de otras funciones, y tienen acceso a cualquier variable que esté en el ámbito en el que se definen. Esto significa que las funciones de JavaScript son *cierres*, y permite importantes y poderosas técnicas de programación.

## 8.1 Definición de funciones

La forma más sencilla de definir una función en JavaScript es con la palabra clave `function`, que puede utilizarse como declaración o como expresión. ES6 define una nueva e importante manera de definir funciones sin la palabra clave `function`: las "funciones flecha" tienen una sintaxis particularmente compacta y son útiles cuando se pasa una función como argumento a otra función. Las subsecciones que siguen cubren estas tres formas de definir funciones. Tenga en cuenta que algunos detalles de la sintaxis de definición de funciones que implican parámetros de función se posponen a §8.3.

En los literales de objeto y en las definiciones de clase, existe una sintaxis abreviada conveniente para definir métodos. Esta sintaxis

abreviada se trató en §6.10.5 y es equivalente a utilizar una expresión de definición de función y asignarla a una propiedad de objeto utilizando la sintaxis básica de literales de objeto name:value. En otro caso especial, puede utilizar las palabras clave get y set en los literales de objeto para definir métodos getter y setter de propiedades especiales. Esta sintaxis de definición de función se trató en §6.10.6.

Tenga en cuenta que las funciones también pueden definirse con el constructor Function(), que es el tema de §8.7.7. Además, JavaScript define algunos tipos especializados de funciones. function\* define funciones generadoras (véase el capítulo 12) y async function define funciones asíncronas (véase el capítulo 13).

### 8.1.1 Declaraciones de funciones

Las declaraciones de funciones constan de la palabra clave function, seguida de estos componentes:

- Un identificador que da nombre a la función. El nombre es una parte obligatoria de las declaraciones de función: se utiliza como nombre de una variable, y el objeto de función recién definido se asigna a la variable.
- Un par de paréntesis alrededor de una lista separada por comas de cero o más identificadores. Estos identificadores son los nombres de los parámetros de la función y se comportan como variables locales dentro del cuerpo de la función.
- Un par de llaves con cero o más sentencias JavaScript en su interior. Estas sentencias son el cuerpo de la función: se ejecutan cada vez que se invoca la función.

Aquí hay algunos ejemplos de declaraciones de funciones:

```
// Imprime el nombre y el valor de cada propiedad de o. Devuelve undefined.
function printprops(o) { for(let p in o) { console.log(`${p}: ${o[p]}\n`); }

}

// Calcular la distancia entre los puntos cartesianos (x1,y1) y
(x2,y2). function distance(x1, y1, x2, y2) { let dx = x2
- x1; let dy = y2 - y1; return Math. sqrt(dx*dx +
dy*dy); }

// Una función recursiva (que se llama a sí misma) que calcula los factoriales //
Recordemos que x! es el producto de x por todos los enteros positivos menores que él.
function factorial(x) { if (x <= 1) return 1; return x * factorial(x-1);
}
```

Una de las cosas importantes que hay que entender sobre las declaraciones de función es que el nombre de la función se convierte en una variable cuyo valor es la propia función. Las declaraciones de funciones se "elevan" a la parte superior del script, la función o el bloque que las encierra, de modo que las funciones definidas de este modo pueden ser invocadas desde el código que aparece antes de la definición. Otra forma de decir esto es que todas las funciones declaradas en un bloque de código JavaScript se definirán a lo largo de ese bloque, y se definirán antes de que el intérprete de JavaScript comience a ejecutar cualquier código de ese bloque.

Las funciones distancia() y factorial() que hemos descrito están diseñadas para calcular un valor, y utilizan return para devolver ese valor a su llamador. La sentencia return hace que la función deje de ejecutarse y devuelva el valor de su expresión (si la hay) a quien la llama. Si la sentencia return no tiene una expresión asociada, el valor de retorno de la función es indefinido.

La función printprops() es diferente: su trabajo consiste en mostrar los nombres y valores de las propiedades de un objeto. No es necesario un valor de retorno, y la función no incluye una declaración de retorno. El valor de una invocación de la función printprops() es siempre indefinido. Si una función no contiene una sentencia de retorno, simplemente ejecuta cada sentencia en el cuerpo de la función hasta que llega al final, y devuelve el valor indefinido a quien la llamó.

Antes de ES6, las declaraciones de funciones sólo se permitían en el nivel superior dentro de un archivo JavaScript o dentro de otra función. Si bien algunas implementaciones se saltaron la regla, no era técnicamente legal definir funciones dentro del cuerpo de los bucles, condicionales u otros bloques. Sin embargo, en el modo estricto de ES6, las declaraciones de funciones están permitidas dentro de los bloques. Sin embargo, una función definida dentro de un bloque sólo existe dentro de ese bloque y no es visible fuera de él.

### 8.1.2 Expresiones de función

Las expresiones de función se parecen mucho a las declaraciones de función, pero aparecen dentro del contexto de una expresión o

declaración mayor, y el nombre es opcional. He aquí algunos ejemplos de expresiones de función:

```
// Esta expresión de función define una función que eleva al cuadrado su argumento. // Observa que lo asignamos a una variable const square = function(x) { return x*x; };
```

// Las expresiones de función pueden incluir nombres, lo cual es útil para la recursión.

```
const f = function fact(x) { if (x <= 1) return 1; else return x*fact(x-1); };
```

// Las expresiones de la función también pueden usarse como argumentos de otras funciones: [3,2,1]. sort(function(a,b) { return a-b; });

// Las expresiones de función se definen a veces y se invocan inmediatamente:  
let tensquared = (function(x) {return x\*x;})(10);

Tenga en cuenta que el nombre de la función es opcional para las funciones definidas como expresiones, y la mayoría de las expresiones de función anteriores que hemos mostrado lo omiten. Una declaración de función *declara* realmente una variable y le asigna un objeto de función. Una expresión de función, en cambio, no declara una variable: es usted quien debe asignar el objeto de función recién definido a una constante o variable si va a necesitar referirse a él varias veces. Es una buena práctica utilizar const con expresiones de función para no sobrescribir accidentalmente las funciones asignando nuevos valores.

Se permite un nombre para las funciones, como la función factorial, que necesitan referirse a sí mismas. Si una expresión de función incluye un nombre, el ámbito de la función local para esa función incluirá un enlace de ese nombre al objeto de la función. En efecto, el nombre de la función se convierte en una variable local dentro de la función. La mayoría de las funciones definidas

como expresiones no necesitan nombres, lo que hace que su definición sea más compacta (aunque no tanto como las funciones de flecha, que se describen a continuación).

Hay una diferencia importante entre definir una función `f()` con una declaración de función y asignar una función a la variable `f` después de crearla como expresión. Cuando se utiliza la forma de declaración, los objetos de la función se crean antes de que el código que los contiene comience a ejecutarse, y las definiciones se elevan para que se pueda llamar a estas funciones desde el código que aparece sobre la declaración de definición. Sin embargo, esto no es cierto para las funciones definidas como expresiones: estas funciones no existen hasta que la expresión que las define se evalúa realmente. Además, para invocar una función, hay que poder referirse a ella, y no se puede referir a una función definida como expresión hasta que se asigne a una variable, por lo que las funciones definidas con expresiones no pueden invocarse antes de ser definidas.

### 8.1.3 Funciones de las flechas

En ES6, se pueden definir funciones utilizando una sintaxis particularmente compacta conocida como "funciones de flecha". Esta sintaxis recuerda a la notación matemática y utiliza una "flecha" `=>` para separar los parámetros de la función del cuerpo de la misma. No se utiliza la palabra clave `function` y, dado que las funciones en flecha son expresiones en lugar de sentencias, tampoco es necesario un nombre de función. La forma general de una función de flecha es una lista de parámetros separada por

comas entre paréntesis, seguida de la flecha =>, seguida del cuerpo de la función entre llaves:

```
const sum = (x, y) => { return x + y; };
```

Pero las funciones de flecha admiten una sintaxis aún más compacta. Si el cuerpo de la función es una única sentencia de retorno, puede omitir la palabra clave return, el punto y coma que la acompaña y las llaves, y escribir el cuerpo de la función como la expresión cuyo valor se va a devolver:

```
const sum = (x, y) => x + y;
```

Además, si una función de flecha tiene exactamente un parámetro, puede omitir los paréntesis alrededor de la lista de parámetros:

```
const polinomio = x => x*x + 2*x + 3;
```

Tenga en cuenta, sin embargo, que una función de flecha sin argumentos debe escribirse con un par de paréntesis vacío:

```
constantFunc = () => 42;
```

Tenga en cuenta que, al escribir una función de flecha, no debe poner una nueva línea entre los parámetros de la función y la flecha =>. De lo contrario, podría terminar con una línea como const polinomio = x, que es una sentencia de asignación sintácticamente válida por sí misma.

Además, si el cuerpo de su función de flecha es una única sentencia de retorno pero la expresión a devolver es un literal de objeto, entonces tiene que poner el literal de objeto dentro de

paréntesis para evitar la ambigüedad sintáctica entre las llaves de un cuerpo de función y las llaves de un literal de objeto:

```
const f = x => { devuelve { valor: x }; }; // Bueno: f() devuelve un objeto const g = x =>
({ valor: x }); // Bueno: g() devuelve un objeto const h = x => { valor: x }; // Malo: h()
no devuelve nada
const i = x => { v: x, w: x }; // Mal: Error de sintaxis
```

En la tercera línea de este código, la función h() es realmente ambigua: el código que pretendía ser un literal de objeto puede ser analizado como una sentencia etiquetada, por lo que se crea una función que devuelve undefined. En la cuarta línea, sin embargo, el literal de objeto más complicado no es una sentencia válida, y este código ilegal provoca un error de sintaxis.

La sintaxis concisa de las funciones de flecha las hace ideales cuando se necesita pasar una función a otra función, lo cual es algo común de hacer con métodos de matrices como map(), filter() y reduce() (ver §7.8.1), por ejemplo:

```
// Hacer una copia de un array con elementos nulos eliminados. let filtered
= [1,null,2,3]. filter(x => x !== null); // filtered == [1,2,3] // Cuadrar algunos
números: let squares = [1,2,3,4]. map(x => x*x); // squares == [1,4,9,16]
```

Las funciones flecha se diferencian de las funciones definidas de otras maneras en un aspecto crítico: heredan el valor de la palabra clave this del entorno en el que se definen en lugar de definir su propio contexto de invocación como hacen las funciones definidas de otras maneras. Esta es una característica importante y muy útil de las funciones de flecha, y volveremos a ella más adelante en este capítulo. Las funciones flecha también se diferencian de otras funciones en que no tienen una propiedad de prototipo, lo que

significa que no pueden utilizarse como funciones constructoras de nuevas clases (véase §9.2).

### 8.1.4 Funciones anidadas

En JavaScript, las funciones pueden estar anidadas dentro de otras funciones. Por ejemplo:

```
función hipotenusa(a, b) {  
  
    function square(x) { return x*x; } return Math.  
    sqrt(square(a) + square(b)); }
```

Lo interesante de las funciones anidadas son sus reglas de alcance de las variables: pueden acceder a los parámetros y variables de la función (o funciones) en la que están anidadas. En el código mostrado aquí, por ejemplo, la función interna `square()` puede leer y escribir los parámetros `a` y `b` definidos por la función externa `hypotenuse()`. Estas reglas de alcance para las funciones anidadas son muy importantes, y las volveremos a considerar en §8.6.

## 8.2 Invocación de funciones

El código JavaScript que compone el cuerpo de una función no se ejecuta cuando se define la función, sino cuando se invoca.

Las funciones de JavaScript pueden ser invocadas de cinco maneras:

- Como funciones
- Como métodos
- Como constructores
- Indirectamente a través de sus métodos call() y apply()
- Implícitamente, a través de las características del lenguaje JavaScript que no aparecen como invocaciones de funciones normales

### 8.2.1 Invocación de funciones

Las funciones se invocan como funciones o como métodos con una expresión de invocación ([§4.5](#)). Una expresión de invocación consiste en una expresión de función que se evalúa a un objeto de función, seguida de un paréntesis de apertura, una lista separada por comas de cero o más expresiones de argumentos y un paréntesis de cierre. Si la expresión de la función es una expresión de acceso a una propiedad -si la función es la propiedad de un objeto o un elemento de un array- entonces es una expresión de invocación a un método. Este caso se explicará en el siguiente ejemplo. El siguiente código incluye una serie de expresiones regulares de invocación de funciones:

```
printprops({x: 1}); let total = distancia(0,0,2,1) + distancia(2,1,3,5);
let probability = factorial(5)/factorial(13);
```

En una invocación, se evalúa cada una de las expresiones de los argumentos (las que están entre paréntesis) y los valores resultantes se convierten en los argumentos de la función. Estos valores se asignan a los parámetros nombrados en la definición de la función. En el cuerpo de la función, una referencia a un parámetro se evalúa al valor del argumento correspondiente.

Para la invocación regular de funciones, el valor de retorno de la función se convierte en el valor de la expresión de invocación. Si la función retorna porque el intérprete llega al final, el valor de retorno es indefinido. Si la función retorna porque el intérprete ejecuta una sentencia de retorno, entonces el valor de retorno es el valor de la expresión que sigue al retorno o es indefinido si la sentencia de retorno no tiene valor.

### INVOCACIÓN CONDICIONAL

En ES2020 puede insertar `?.` después de la expresión de la función y antes del paréntesis abierto en un para invocar la función sólo si no es `null` o `indefinido`. Es decir, el expresión `f?.(x)` es equivalente (asumiendo que no hay efectos

```
(f !== null && f !== undefined) ? f(x) : undefined
```

Los detalles completos de esta sintaxis de invocación [§4.5.1](#) se

Para la invocación de funciones en modo no estricto, el contexto de invocación (el valor `this`) es el objeto global. En modo estricto, sin embargo, el contexto de invocación es indefinido. Tenga en cuenta que las funciones definidas mediante la sintaxis de flecha se comportan de forma diferente: siempre heredan el valor de `this` que está en vigor en el lugar donde se definen.

Las funciones escritas para ser invocadas como funciones (y no como métodos) no suelen utilizar la palabra clave `this`. Sin embargo, la palabra clave puede utilizarse para determinar si el modo estricto está en vigor:

```
// Definir e invocar una función para determinar si estamos en modo
estricto. const strict = (function() { return ! this; }());
```

Una función *recursiva* es aquella, como la función factorial() del principio de este capítulo, que se llama a sí misma. Algunos algoritmos, como los que implican estructuras de datos basadas en árboles, pueden implementarse de forma particularmente elegante con funciones recursivas. Sin embargo, al escribir una función recursiva, es importante pensar en las restricciones de memoria. Cuando una función A llama a la función B, y luego la función B llama a la función C, el intérprete de JavaScript tiene que hacer un seguimiento de los contextos de ejecución de las tres funciones. Cuando la función C termina, el intérprete necesita saber dónde reanudar la ejecución de la función B, y cuando la función B termina, necesita saber dónde reanudar la ejecución de la función A. Puedes imaginar estos contextos de ejecución como una pila. Cuando una función llama a otra función, un nuevo contexto de ejecución es introducido en la pila. Cuando esa función regresa, su objeto de contexto de ejecución es retirado de la pila. Si una función se llama a sí misma recursivamente 100 veces, la pila tendrá 100 objetos empujados en ella, y luego esos 100 objetos serán retirados. Esta pila de llamadas requiere memoria. En el hardware moderno, normalmente está bien escribir funciones recursivas que se llamen a sí mismas cientos de veces. Pero si una función se llama a sí misma diez mil veces, es probable que falle con un error como "Maximum call-stack size exceeded".

## 8.2.2 Invocación de métodos

Un *método* no es más que una función de JavaScript que se almacena en una propiedad de un objeto. Si tienes una función f y un objeto o, puedes definir un método llamado m de o con la siguiente línea:

```
o. m = f;
```

Habiendo definido el método m() del objeto o, invócalo así:

```
o. m();
```

O bien, si m() espera dos argumentos, podrías invocarla así:

```
o. m(x, y);
```

El código de este ejemplo es una expresión de invocación: incluye una expresión de función o.m y dos expresiones de argumento, x e y. La expresión de función es en sí misma una expresión de acceso a propiedades, y esto significa que la función se invoca como un método en lugar de como una función regular.

Los argumentos y el valor de retorno de una invocación de método se manejan exactamente como se describe para la invocación de una función regular. Sin embargo, las invocaciones a métodos difieren de las invocaciones a funciones en un aspecto importante: el contexto de invocación. Las expresiones de acceso a propiedades constan de dos partes: un objeto (en este caso o) y un nombre de propiedad (m). En una expresión de invocación de método como ésta, el objeto o se convierte en el contexto de invocación, y el cuerpo de la función puede referirse a ese objeto utilizando la palabra clave this. He aquí un ejemplo concreto:

```
let calculator = { // Un literal de objeto operando1: 1, operando2: 1, add() { // Estamos usando la sintaxis de método abreviado para esta función // Nótese el uso de la palabra clave this para referirse al objeto contenedor.  
    this. result = this. operand1 + this. operand2; } }; calculator. add(); // Una invocación al método para calcular 1+1.  
calculadora. resultado //=> 2
```

La mayoría de las invocaciones de métodos utilizan la notación de puntos para el acceso a propiedades, pero las expresiones de acceso a propiedades que utilizan corchetes también provocan la invocación de métodos. Las siguientes son ambas invocaciones de métodos, por ejemplo:

```
o["m"](x,y); // Otra forma de escribir o.m(x,y).  
a[0](z) // También una invocación a un método (asumiendo que a[0] es una función).
```

Las invocaciones de métodos también pueden implicar expresiones de acceso a propiedades más complejas:

```
customer.surname.toUpperCase(); // Invocar el método sobre  
customer.surname f(). m(); // Invocar el método m() sobre el valor de  
retorno de f()
```

Los métodos y la palabra clave `this` son fundamentales para el paradigma de la programación orientada a objetos. A cualquier función que se utilice como método se le pasa un argumento implícito: el objeto a través del cual se invoca. Normalmente, un método realiza algún tipo de operación sobre ese objeto, y la sintaxis de invocación de métodos es una forma elegante de expresar el hecho de que una función está operando sobre un objeto. Compare las dos líneas siguientes:

```
rect.setSize(width, height); setRectSize(rect, width, height);
```

Las hipotéticas funciones invocadas en estas dos líneas de código pueden realizar exactamente la misma operación sobre el (hipotético) objeto `rect`, pero la sintaxis de invocación de métodos en la primera línea indica más claramente la idea de que es el objeto `rect` el principal foco de la operación.

## ENCADENAMIENTO DE

Cuando devuelven objetos, puede utilizar el valor de retorno de una invocación de método como parte de un la siguiente invocación. Esto da lugar a una serie (o "cadena") de invocaciones de métodos como un único expresión. Cuando se trabaja con operaciones asíncronas basadas en promesas [Capítulo 43](#), Por ejemplo, es común escribir código estructurado de esta manera:

```
// Ejecuta tres operaciones asíncronas en secuencia, manejando los errores.  
doStepOne().ento(doStepTwo).ento(doStepThree).atrap(handleErrors);
```

Cuando escriba un método que no tenga un valor de retorno propio, considere la posibilidad de que el método devolverá este. Si haces esto de forma consistente en toda tu API, habilitarás un estilo de programación conocido encadenamiento de en el que un objeto puede ser nombrado una vez y luego múltiples métodos pueden ser invocada en ella:

```
nuev Cuadra().x(100).y(100).tam(50).esboza("rojo").llena("azul").dibuj();
```

Tenga en cuenta que se trata de una palabra clave, no de un nombre de variable o propiedad.

La sintaxis de JavaScript no permite asignar un valor a esto.

La palabra clave this no tiene el mismo alcance que las variables y, excepto en el caso de las funciones de flecha, las funciones anidadas no heredan el valor this de la función que las contiene. Si una función anidada es invocada como un método, su valor this es el objeto sobre el que fue invocada. Si una función anidada (que no es una función de flecha) es invocada como una función, entonces su valor this será el objeto global (modo no estricto) o indefinido (modo estricto). Es un error común asumir que una función anidada definida dentro de un método e invocada como una función puede usar esto para obtener el contexto de invocación del método. El siguiente código demuestra el problema:

```

let o = { // Un objeto o. m: function() { // Método m del objeto.
    let self = this; // Guardar el valor de "this" en una variable.
    this === o // => true: "this" es el objeto o. f(); // Ahora llama a la función de ayuda
    f().
        function f() { // Una función anidada f this === o // => false: "this" es global o
indefinido self === o // => true: self es el valor externo de "this".
    }
} };
o. m(); // Invocar el método m en el objeto o.

```

Dentro de la función anidada f(), la palabra clave this no es igual al objeto o. Esto es ampliamente considerado como un defecto en el lenguaje JavaScript, y es importante ser consciente de ello. El código anterior demuestra una solución común. Dentro del método m, asignamos el valor de this a una variable self, y dentro de la función anidada f, podemos usar self en lugar de this para referirnos al objeto contenedor.

En ES6 y posteriores, otra solución a este problema es convertir la función anidada f en una función de flecha, que heredará correctamente este valor:

```

const f = () => {
    this === o // true, ya que las funciones de flecha heredan esto;
}

```

Las funciones definidas como expresiones en lugar de sentencias no se izan, por lo que para que este código funcione, la definición de la función para f tendrá que moverse dentro del método m para que aparezca antes de ser invocado.

Otra solución es invocar el método bind() de la función anidada para definir una nueva función que se invoca implícitamente sobre un objeto especificado:

```
const f = (function() { this === o // true, ya que hemos vinculado esta función al exterior this }).bind(this);
```

Hablaremos más sobre bind() en [§8.7.5](#).

### 8.2.3 Invocación del constructor

Si una invocación a una función o método está precedida por la palabra clave new, entonces es una invocación a un constructor. (Las invocaciones a constructores se introdujeron en [§4.6](#) y [§6.2.2](#), y los constructores se tratarán con más detalle en el [capítulo 9](#)).

Las invocaciones a constructores difieren de las invocaciones a funciones y métodos regulares en su manejo de argumentos, contexto de invocación y valor de retorno.

Si una invocación a un constructor incluye una lista de argumentos entre paréntesis, esas expresiones de argumentos se evalúan y se pasan a la función de la misma manera que lo harían para las invocaciones de funciones y métodos. No es una práctica común, pero puede omitir un par de paréntesis vacíos en una invocación a un constructor. Las dos líneas siguientes, por ejemplo, son equivalentes:

```
o = new Object(); o =  
new Object;
```

Una invocación a un constructor crea un nuevo objeto vacío que hereda del objeto especificado por la propiedad prototype del

constructor. Las funciones constructoras están pensadas para inicializar objetos, y este objeto recién creado se utiliza como contexto de invocación, por lo que la función constructora puede referirse a él con la palabra clave `this`. Tenga en cuenta que el nuevo objeto se utiliza como contexto de invocación incluso si la invocación del constructor parece una invocación a un método. Es decir, en la expresión `new o.m()`, o no se utiliza como contexto de invocación.

Las funciones constructoras no suelen utilizar la palabra clave `return`. Normalmente inicializan el nuevo objeto y luego retornan implícitamente cuando llegan al final de su cuerpo. En este caso, el nuevo objeto es el valor de la expresión de invocación del constructor. Sin embargo, si un constructor utiliza explícitamente la sentencia `return` para devolver un objeto, entonces ese objeto se convierte en el valor de la expresión de invocación. Si el constructor utiliza `return` sin valor, o si devuelve un valor primitivo, ese valor de retorno se ignora y el nuevo objeto se utiliza como valor de la invocación.

#### 8.2.4 Invocación indirecta

Las funciones de JavaScript son objetos, y como todos los objetos de JavaScript, tienen métodos. Dos de estos métodos, `call()` y `apply()`, invocan la función indirectamente. Ambos métodos permiten especificar explícitamente el valor `this` para la invocación, lo que significa que se puede invocar cualquier función como método de cualquier objeto, aunque no sea realmente un método de ese objeto. Ambos métodos también permiten especificar los argumentos para la invocación. El método `call()` utiliza su propia lista de argumentos como argumentos de la

función, y el método `apply()` espera que se utilice una matriz de valores como argumentos. Los métodos `call()` y `apply()` se describen en detalle en [§8.7.4](#).

### 8.2.5 Invocación implícita de funciones

Hay varias características del lenguaje JavaScript que no parecen invocaciones de funciones pero que hacen que se invoquen funciones. Tenga mucho cuidado cuando escriba funciones que puedan ser invocadas implícitamente, porque los errores, los efectos secundarios y los problemas de rendimiento en estas funciones son más difíciles de diagnosticar y arreglar que en las funciones normales, por la sencilla razón de que puede no ser obvio a partir de una simple inspección de su código cuándo están siendo llamadas.

Las características del lenguaje que pueden causar la invocación implícita de funciones incluyen:

- Si un objeto tiene definidos getters o setters, la consulta o el establecimiento del valor de sus propiedades puede invocar esos métodos. Véase [§6.10.6](#) para más información.
- Cuando un objeto se utiliza en un contexto de cadena (como cuando se concatena con una cadena), se llama a su método `toString()`. Del mismo modo, cuando un objeto se utiliza en un contexto numérico, se invoca su método `valueOf()`. Para más detalles, véase [§3.9.3](#).
- Cuando se hace un bucle sobre los elementos de un objeto iterable, hay una serie de llamadas a métodos que se producen. El [capítulo 12](#) explica cómo funcionan los iteradores a nivel de llamadas a funciones y demuestra

cómo escribir estos métodos para que puedas definir tus propios tipos de iterables.

- Un literal de plantilla etiquetado es una invocación a una función disfrazada. En §14.5 se muestra cómo escribir funciones que pueden usarse junto con cadenas de literales de plantilla.
- Los objetos proxy (descritos en §14.7) tienen su comportamiento completamente controlado por funciones. Casi cualquier operación sobre uno de estos objetos hará que se invoque una función.

## 8.3 Argumentos y parámetros de las funciones

Las definiciones de funciones de JavaScript no especifican un tipo esperado para los parámetros de la función, y las invocaciones de funciones no hacen ninguna comprobación de tipo en los valores de los argumentos que se pasan. De hecho, las invocaciones de funciones de JavaScript ni siquiera comprueban el número de argumentos que se pasan. Las subsecciones siguientes describen lo que ocurre cuando se invoca una función con menos argumentos que los parámetros declarados o con más argumentos que los parámetros declarados. También demuestran cómo puede comprobar explícitamente el tipo de argumentos de la función si necesita asegurarse de que una función no se invoca con argumentos inapropiados.

### 8.3.1 Parámetrosopcionales y valores por defecto

Cuando una función es invocada con menos argumentos que los parámetros declarados, los parámetros adicionales se fijan a su valor por defecto, que normalmente es indefinido. A menudo

resulta útil escribir las funciones de forma que algunos argumentos sean opcionales. El siguiente es un ejemplo:

```
// Añade los nombres de las propiedades enumerables del objeto o a la matriz a,  
y devuelve a. Si se omite a, crea y devuelve una nueva matriz.  
function getPropertyNames(o, a) { if (a === undefined) a = []; // Si es undefined, usa  
un nuevo array for(let property in o) a.push(property); return a; }  
  
// getPropertyNames() puede ser invocado con uno o dos argumentos:  
let o = {x: 1}, p = {y: 2, z: 3}; // Dos objetos de prueba let a = getPropertyNames(o);  
// a == ["x"]; obtener las propiedades de o en un nuevo array getPropertyNames(p,  
a); // a == ["x", "y", "z"]; añadirle las propiedades de p
```

En lugar de utilizar una sentencia if en la primera línea de esta función, puede utilizar el operador || de esta forma idiomática:

```
a = a || [];
```

Recuerde de [§4.10.2](#) que el operador || devuelve su primer argumento si éste es verdadero y, en caso contrario, devuelve su segundo argumento. En este caso, si se pasa algún objeto como segundo argumento, la función utilizará ese objeto. Pero si se omite el segundo argumento (o se pasa null u otro valor falso), se utilizará un array vacío recién creado.

Tenga en cuenta que cuando diseñe funciones con argumentosopcionales, debe asegurarse de poner los opcionales al final de la lista de argumentos para que puedan ser omitidos. El programador que llama a su función no puede omitir el primer argumento y pasar el segundo: tendría que pasar explícitamente undefined como primer argumento.

En ES6 y posteriores, puede definir un valor por defecto para cada uno de los parámetros de su función directamente en la lista de

parámetros de su función. Simplemente sigue el nombre del parámetro con un signo igual y el valor por defecto que se utilizará cuando no se proporcione ningún argumento para ese parámetro:

```
// Añade los nombres de las propiedades enumerables del objeto o a la matriz a, y devuelve a. Si se omite a, crea y devuelve una nueva matriz.  
function getPropertyNames(o, a = []) { for(let property in o) a.  
push(property); return a;  
}
```

Las expresiones de parámetros por defecto se evalúan cuando se llama a la función, no cuando se define, por lo que cada vez que esta

La función `getPropertyNames()` es invocada con un argumento, un nuevo array vacío es creado y pasado.<sup>2</sup> Probablemente sea más fácil razonar sobre las funciones si los parámetros por defecto son constantes (o expresiones literales como `[]` y `{}`). Pero esto no es necesario: se pueden utilizar variables, o invocaciones de funciones, por ejemplo, para calcular el valor por defecto de un parámetro. Un caso interesante es que, en el caso de las funciones con múltiples parámetros, se puede utilizar el valor de un parámetro anterior para definir el valor por defecto de los parámetros que le siguen:

```
// Esta función devuelve un objeto que representa un rectángulo dimensions. // Si sólo se suministra la anchura, hazlo el doble de alto que de ancho.  
const rectangle = (width, height=width*2) => ({width, height}); rectangle(1)  
// => { width: 1, height: 2 }
```

Este código demuestra que los parámetros por defecto funcionan con las funciones de flecha. Lo mismo ocurre con las funciones de

método abreviado y todas las demás formas de definición de funciones.

### 8.3.2 Parámetros de reposo y listas de argumentos de longitud variable

Los parámetros por defecto nos permiten escribir funciones que pueden ser invocadas con menos argumentos que parámetros. Los parámetros de *reposo* permiten el caso contrario: nos permiten escribir funciones que pueden ser invocadas con un número arbitrario de argumentos que los parámetros. He aquí un ejemplo de función que espera uno o más argumentos numéricos y devuelve el mayor de ellos:

```
function max(first=-Infinity, ... rest) { let maxValue = first; // Empieza asumiendo que el primer argumento es el mayor  
    // A continuación, se realiza un bucle por el resto de los argumentos, buscando los más grandes for(let n of rest) { if (n > maxValue) { maxValue = n; }  
    } // Devuelve el mayor return maxValue; } max(1, 10, 100, 2, 3,  
1000, 4, 5, 6) //=> 1000
```

Un parámetro de reposo va precedido de tres puntos y debe ser el último parámetro de una declaración de función. Cuando se invoca una función con un parámetro restante, los argumentos que se pasan se asignan primero a los parámetros no restantes, y luego los argumentos restantes (es decir, el "resto" de los argumentos) se almacenan en una matriz que se convierte en el valor del parámetro restante. Este último punto es importante: dentro del

cuerpo de una función, el valor de un parámetro de descanso siempre será un array. La matriz puede estar vacía, pero un parámetro restante nunca estará indefinido. (De esto se deduce que nunca es útil -y no es legal- definir un parámetro por defecto para un parámetro de reposo).

Las funciones como el ejemplo anterior que pueden aceptar cualquier número de argumentos se llaman *funciones variádicas*, *funciones de aridad variable* o *funciones vararg*. En este libro se utiliza el término más coloquial, *varargs*, que data de los primeros días del lenguaje de programación C.

No confunda el ... que define un parámetro de reposo en una definición de función con el operador de propagación ..., descrito en §8.3.4, que puede utilizarse en las invocaciones de funciones.

### 8.3.3 El objeto Arguments

Los parámetros Rest se introdujeron en JavaScript en ES6. Antes de esa versión del lenguaje, las funciones varargs se escribían utilizando el objeto Arguments: dentro del cuerpo de cualquier función, el identificador arguments hace referencia al objeto Arguments para esa invocación. El objeto Arguments es un objeto tipo array (véase §7.9) que permite recuperar los valores de los argumentos pasados a la función por número, en lugar de por nombre. Aquí está la función max() de antes, reescrita para utilizar el objeto Arguments en lugar de un parámetro de reposo:

```
function max(x) { let maxValue = -Infinity;  
    // Recorre los argumentos, buscando y recordando el más grande.  
    for(let i = 0; i < arguments.length; i++) { if (arguments[i] > maxValue) maxValue =  
        arguments[i];  
    } // Devuelve el mayor return maxValue; } max(1, 10, 100, 2, 3,  
1000, 4, 5, 6) //=> 1000
```

El objeto Arguments se remonta a los primeros días de JavaScript y lleva consigo un extraño bagaje histórico que lo hace ineficiente y difícil de optimizar, especialmente fuera del modo estricto. Es posible que aún encuentres código que utilice el objeto Arguments, pero deberías evitar utilizarlo en cualquier código nuevo que escribas. Al refactorizar código antiguo, si encuentra una función que utiliza argumentos, a menudo puede reemplazarla con un parámetro de reposo ...args. Parte del desafortunado legado del objeto Arguments es que, en modo estricto, arguments se trata como una palabra reservada, y no puedes declarar un parámetro de función o una variable local con ese nombre.

### 8.3.4 El operador Spread para las llamadas a funciones

El operador de propagación ... se utiliza para desempaquetar, o "repartir", los elementos de un array (o cualquier otro objeto iterable, como las cadenas) en un contexto en el que se esperan valores individuales. En [§7.1.2 hemos](#) visto cómo se utiliza el operador de dispersión con los literales de las matrices. El operador puede utilizarse, del mismo modo, en las invocaciones de funciones:

```
let numbers = [5, 2, 10, -1, 9, 100, 1]; Math.min(... numbers) //=> -1
```

Tenga en cuenta que ... no es un operador verdadero en el sentido de que no puede ser evaluado para producir un valor. En cambio, es una sintaxis especial de JavaScript que puede utilizarse en los literales de matrices y en las invocaciones de funciones.

Cuando utilizamos la misma sintaxis ... en una definición de función en lugar de una invocación de función, tiene el efecto contrario al del operador spread. Como vimos en §8.3.2, el uso de ... en una definición de función reúne múltiples argumentos de la función en un array. Los parámetros de reposo y el operador de propagación son a menudo útiles juntos, como en la siguiente función, que toma un argumento de función y devuelve una versión instrumentada de la función para su comprobación:

```
// Esta función toma una función y devuelve una versión envuelta function timed(f) {  
    return function(... args) { // Recoge los args en una matriz de parámetros de reposo  
        console.log(`Entrando función ${f.name}`); let startTime = Date.now(); try {  
            // Pasa todos nuestros argumentos a la función envuelta return f(... args); //  
            // Vuelve a repartir los argumentos } finally {  
            // Antes de devolver el valor de retorno envuelto, imprime el tiempo  
            // transcurrido.  
            console.log(`Exiting ${f.name} after ${Date.now()-startTime}ms`);  
        }  
    };  
}
```

```
// Calcula la suma de los números entre 1 y n por fuerza bruta función  
benchmark(n) { deja que sum = 0; for(deja que i = 1; i <= n; i++) sum += i; return  
sum; }
```

```
// Ahora invoca la versión temporizada de esa función de prueba  
timed(benchmark)(1000000) // => 500000500000; esta es la suma de los números
```

### 8.3.5 Desestructuración de los argumentos de la función en parámetros

Cuando se invoca una función con una lista de valores de argumentos, esos valores acaban siendo asignados a los parámetros declarados en la definición de la función. Esta fase inicial de la invocación de la función se parece mucho a la asignación de variables. Así que no debería sorprender que podamos utilizar las técnicas de asignación de desestructuración (véase §3.10.3) con las funciones.

Si define una función que tiene nombres de parámetros entre corchetes, le está diciendo a la función que espere que se pase un valor de matriz por cada par de corchetes. Como parte del proceso de invocación, los argumentos de la matriz se descompondrán en los parámetros con nombres individuales. Como ejemplo, supongamos que estamos representando vectores 2D como arrays de dos números, donde el primer elemento es la coordenada X y el segundo elemento es la coordenada Y. Con esta sencilla estructura de datos, podríamos escribir la siguiente función para sumar dos vectores:

```
function vectorAdd(v1, v2) { return [v1[0] + v2[0], v1[1]
+ v2[1]]; }
vectorAdd([1,2], [3,4]) //=> [4,6]
```

El código sería más fácil de entender si desestructuráramos los dos argumentos vectoriales en parámetros con nombres más claros:

```
function vectorAdd([x1,y1], [x2,y2]) { // Descompone 2 argumentos en 4
parámetros return [x1 + x2, y1 + y2]; }
vectorAdd([1,2], [3,4]) //=> [4,6]
```

Del mismo modo, si está definiendo una función que espera un argumento de objeto, puede desestructurar los parámetros de ese objeto. Volvamos a utilizar un ejemplo de vector, sólo que esta vez supongamos que representamos los vectores como objetos con parámetros `x` e `y`:

```
// Multiplica el vector {x,y} por un valor escalar function
vectorMultiply({x, y}, scalar) { return { x: x*scalar, y: y*scalar }; }
vectorMultiply({x: 1, y: 2}, 2) //=> {x: 2, y: 4}
```

Este ejemplo de desestructuración de un único argumento de objeto en dos parámetros es bastante claro porque los nombres de los parámetros que utilizamos coinciden con los nombres de las propiedades del objeto entrante. La sintaxis es más verbosa y más confusa cuando se necesita desestructurar propiedades con un nombre en parámetros con nombres diferentes. Este es el ejemplo de adición de vectores, implementado para vectores basados en objetos:

```
función vectorAdd(
  {x: x1, y: y1}, // Desempaquetar el primer objeto en los parámetros x1 e y1
  {x: x2, y: y2} // Descomponer el segundo objeto en x2 e y2
  params ) { return { x: x1 + x2, y: y1 + y2 }; }
vectorAdd({x: 1, y: 2}, {x: 3, y: 4}) //=> {x: 4, y: 6}
```

Lo difícil de la sintaxis de desestructuración como `{x:x1, y:y1}` es recordar cuáles son los nombres de las propiedades y cuáles los de los parámetros. La regla a tener en cuenta para las asignaciones de desestructuración y las llamadas a funciones de desestructuración es que las variables o los parámetros que se declaran van en los lugares donde se espera que vayan los valores en un literal de objeto. Por lo tanto, los nombres de las propiedades van siempre a

la izquierda de los dos puntos, y los nombres de los parámetros (o variables) a la derecha.

Puede definir los parámetros por defecto con parámetros desestructurados.

Aquí está la multiplicación de vectores que funciona con vectores 2D o 3D:

```
// Multiplica el vector {x,y} o {x,y,z} por un valor escalar
function vectorMultiply({x, y, z=0}, scalar) { return { x: x*scalar, y: y*scalar, z: z*scalar }; }
vectorMultiply({x: 1, y: 2}, 2) // => {x: 2, y: 4, z: 0}
```

Algunos lenguajes (como Python) permiten a quien llama a una función invocar una función con argumentos especificados en forma de nombre=valor, lo cual es conveniente cuando hay muchos argumentos opcionales o cuando la lista de parámetros es lo suficientemente larga como para que sea difícil recordar el orden correcto. JavaScript no permite esto directamente, pero se puede aproximar desestructurando un argumento de objeto en los parámetros de la función. Consideremos una función que copia un número determinado de elementos de una matriz en otra matriz con desplazamientos iniciales especificados opcionalmente para cada matriz. Dado que hay cinco parámetros posibles, algunos de los cuales tienen valores por defecto, y que sería difícil para un llamador recordar en qué orden pasar los argumentos, podemos definir e invocar la función arraycopy() así:

```
function arraycopy({de, a=desde, n=longitud dede, fromIndex=0, toIndex=0}) { let
valuesToCopy = from. slice(fromIndex, fromIndex + n); to. splice(toIndex, 0, ...
valuesToCopy); return to; } let a = [1,2,3,4,5], b = [9,8,7,6,5]; arraycopy({from: a,
n: 3, to: b, toIndex: 4}) // => [9,8,7,6,1,2,3,5]
```

Cuando se desestructura un array, se puede definir un parámetro de reposo para valores extra dentro del array que se está desestructurando. Ese parámetro de reposo dentro de los corchetes es completamente diferente al verdadero parámetro de reposo de la función:

```
// Esta función espera un argumento de matriz. Los dos primeros elementos de ese
// array se desempaquetan en los parámetros x e y. Los elementos restantes
// se almacenan en el array de coords. Y cualquier argumento después del primer
// array // se empaqueta en el array de resto.
function f([x, y, ... coords], ... rest) { devuelve [x+y, ... rest, ... coords]; // Nota:
el operador spread aquí } f([1, 2, 3, 4], 5, 6) // => [3, 5, 6, 3, 4]
```

En ES2018, también se puede utilizar un parámetro de reposo cuando se desestructura un objeto. El valor de ese parámetro de reposo será un objeto que tenga cualquier propiedad que no se haya desestructurado. Los parámetros de reposo de objetos suelen ser útiles con el operador de propagación de objetos, que también es una nueva característica de ES2018:

```
// Multiplica el vector {x,y} o {x,y,z} por un valor escalar, conserva otros accesorios
function vectorMultiply({x, y, z=0, ... props}, scalar) { return { x: x*escalar, y:
y*escalar, z: z*escalar, ... props }; } vectorMultiply({x: 1, y: 2, w: -1}, 2) // => {x: 2, y:
4, z: 0, w: -1}
```

Por último, tenga en cuenta que, además de desestructurar objetos y arrays de argumentos, también puede desestructurar arrays de objetos, objetos que tienen propiedades de arrays y objetos que tienen propiedades de objetos, básicamente a cualquier profundidad. Considere un código gráfico que representa círculos como objetos con propiedades x, y, radio y color, donde la propiedad de color es una matriz de componentes de color rojo,

verde y azul. Puede definir una función que espera que se le pase un único objeto círculo, pero que desestructura ese objeto círculo en seis parámetros distintos:

```
function drawCircle({x, y, radius, color: [r, g, b]}) { // Aún no se ha implementado  
}
```

Si la desestructuración de los argumentos de la función es más complicada que esto, encuentro que el código se vuelve más difícil de leer, en lugar de ser más simple. A veces, es más claro ser explícito sobre el acceso a las propiedades de los objetos y la indexación de los arrays.

### 8.3.6 Tipos de argumentos

Los parámetros de los métodos de JavaScript no tienen tipos declarados, y no se realiza ninguna comprobación de tipos en los valores que se pasan a una función. Puede ayudar a que su código sea auto-documentado eligiendo nombres descriptivos para los argumentos de las funciones y documentándolos cuidadosamente en los comentarios de cada función. (Como alternativa, vea [§17.8](#) para una extensión del lenguaje que le permite poner una capa de comprobación de tipos sobre el JavaScript normal).

Como se describe en [§3.9](#), JavaScript realiza una conversión de tipo liberal según sea necesario. Así que si escribes una función que espera un argumento de tipo cadena y luego llamas a esa función con un valor de algún otro tipo, el valor que pasaste simplemente se convertirá en una cadena cuando la función intente usarlo como una cadena. Todos los tipos primitivos pueden ser convertidos a cadenas, y todos los objetos tienen métodos `toString()` (si no son

necesariamente útiles), por lo que nunca se produce un error en este caso.

Sin embargo, esto no siempre es cierto. Considere de nuevo el método `arraycopy()` mostrado anteriormente. Espera uno o dos argumentos del array y fallará si estos argumentos son del tipo incorrecto. A menos que esté escribiendo una función privada que sólo será llamada desde partes cercanas de su código, puede valer la pena añadir código para comprobar los tipos de argumentos como éste. Es mejor que una función falle inmediatamente y de forma predecible cuando se le pasen valores erróneos que empezar a ejecutarse y fallar más tarde con un mensaje de error que probablemente sea poco claro. A continuación se muestra una función de ejemplo que realiza la comprobación de tipos:

```
// Devuelve la suma de los elementos un objeto iterable a.  
// Los elementos de a deben ser todos números.  
function sum(a) { let total =  
  0;  
  
  for(let element of a) { // Throws TypeError si a no es iterable if (typeof element !=  
  "number") {  
    throw new TypeError("sum(): elements must be numbers"); } total +=  
    element; } return total; } sum([1,2,3]) // => 6 sum(1, 2, 3); // !TypeError: 1 no es  
  iterable sum([1,2, "3"]); // !TypeError: el elemento 2 no es un número
```

## 8.4 Funciones como valores

Las características más importantes de las funciones son que pueden ser definidas e invocadas. La definición e invocación de

funciones son características sintácticas de JavaScript y de la mayoría de los demás lenguajes de programación. En JavaScript, sin embargo, las funciones no son sólo sintaxis, sino también valores, lo que significa que pueden asignarse a variables, almacenarse en las propiedades de los objetos o en los elementos de las matrices, pasarse como argumentos a las funciones, etc.<sup>3</sup>

Para entender cómo las funciones pueden ser datos de JavaScript, así como la sintaxis de JavaScript, considere esta definición de función:

```
función cuadrado(x) { devuelve x*x; }
```

Esta definición crea un nuevo objeto función y lo asigna a la variable cuadrado. El nombre de una función es realmente irrelevante; es simplemente el nombre de una variable que se refiere al objeto función. La función puede ser asignada a otra variable y seguir funcionando de la misma manera:

```
let s = square; // Ahora s se refiere a la misma función que square square(4) //  
=> 16 s(4) // => 16
```

Las funciones también se pueden asignar a las propiedades del objeto en lugar de a las variables. Como ya hemos comentado, llamamos a las funciones "métodos" cuando hacemos esto:

```
let o = {square: function(x) { return x*x; }}; // Un objeto literal  
let y = o.square(16); // y == 256
```

Las funciones ni siquiera requieren nombres, como cuando se asignan a los elementos de una matriz:

```
let a = [x => x*x, 20]; // Un literal de matriz a[0](a[1]) // => 400
```

La sintaxis de este último ejemplo parece extraña, pero sigue siendo una expresión de invocación de función legal.

Como ejemplo de la utilidad de tratar las funciones como valores, considere el método `Array.sort()`. Este método ordena los elementos de un array. Dado que hay muchos órdenes posibles para ordenar (orden numérico, orden alfabético, orden de fechas, ascendente, descendente, etc.), el método `sort()` toma opcionalmente una función como argumento para indicarle cómo realizar la ordenación. Esta función tiene un trabajo sencillo: para dos valores cualesquiera que se le pasen, devuelve un valor que especifica qué elemento vendría primero en un array ordenado. Este argumento de la función hace que `Array.sort()` sea perfectamente general e infinitamente flexible; puede ordenar cualquier tipo de datos en cualquier orden concebible. En [§7.8.6](#) se muestran ejemplos.

El ejemplo 8-1 demuestra el tipo de cosas que se pueden hacer cuando se utilizan funciones como valores. Este ejemplo puede ser un poco complicado, pero los comentarios explican lo que sucede.

*Ejemplo 8-1. Uso de funciones como datos*

---

```

// Definimos algunas funciones simples aquí función
sumar(x,y) { devolver x + y; } función restar(x,y) { devolver x -
y; } función multiplicar(x,y) { devolver x * y; } función
dividir(x,y) { devolver x / y; }

// Aquí hay una función que toma una de las funciones anteriores
// como argumento y lo invoca sobre dos operandos function
operate(operator, operand1, operand2) { return operator(operand1,
operand2); }

// Podríamos invocar esta función así para calcular el valor
(2+3) + (4*5):
deje i = operate(add, operate(add, 2, 3), operate(multiply, 4, 5));

// Por el bien del ejemplo, volvemos a implementar las funciones simples, // esta vez
dentro de un literal de objeto; const operators = { add: (x,y) => x+y, restar: (x,y) => x-y,
multiplicar: (x,y) => x*y, dividir: (x,y) => x/y, pow: Math. pow // Esto también
funciona para las funciones predefinidas };

// Esta función toma el nombre de un operador, busca ese operador
// en el objeto, y luego lo invoca sobre los operandos suministrados.
Nota

```

```

// la sintaxis utilizada para invocar la función del operador. function
operate2(operation, operand1, operand2) { if (typeof operators[operation] ===
"function") { return operators[operation](operand1, operand2); } else throw
"unknown operator"; }

operate2("add", "hello", operate2("add", " ", "world")) // => "hola mundo"
operate2("pow", 10, 2) // => 100

```

## 8.4.1 Definir sus propias propiedades de función

.

Las funciones no son valores primitivos en JavaScript, sino un tipo especializado de objeto, lo que significa que las funciones pueden tener propiedades. Cuando una función necesita una variable "estática" cuyo valor persiste a través de las invocaciones, a menudo es conveniente utilizar una propiedad de la propia función. Por ejemplo, supongamos que queremos escribir una función que devuelva un único número entero cada vez que se invoque. La función nunca debe devolver el mismo valor dos veces. Para ello, la función necesita llevar un registro de los valores que ya ha devuelto, y esta información debe persistir a través de las invocaciones de la función. Podrías almacenar esta información en una variable global, pero eso es innecesario, porque la información sólo la utiliza la propia función. Es mejor almacenar la información en una propiedad del objeto Función. Aquí hay un ejemplo que devuelve un entero único cada vez que es llamado:

```
// Inicializar la propiedad contador del objeto función.  
// Las declaraciones de las funciones se elevan para que realmente podamos  
// hacer esta asignación antes de la declaración de la función. uniqueInteger.counter =  
0;  
  
// Esta función devuelve un entero diferente cada vez que se llama.  
// Utiliza una propiedad de sí misma para recordar el siguiente valor a devolver.  
function uniqueInteger() { return uniqueInteger.counter++; // Devuelve e  
incrementa la propiedad del contador } uniqueInteger() // => 0 uniqueInteger() //  
=> 1
```

Como otro ejemplo, considere la siguiente función factorial() que utiliza las propiedades de sí misma (tratándose como una matriz) para almacenar en caché los resultados previamente calculados:

```
// Calcula los factoriales y cachea los resultados como propiedades de la propia función.
function factorial(n) { if (Number. isInteger(n) && n > 0) { // Sólo enteros positivos if (!
(n in factorial)) { // Si no hay resultado en caché factorial[n] = n * factorial(n-1); //
Calcula y lo almacena en caché } return factorial[n]; // Devuelve el resultado en caché }
else { return NaN; // Si la entrada fue mala } } factorial[1] = 1; // Inicializa la caché para
mantener este caso base.
factorial(6) // => 720 factorial[5] // => 120; la llamada anterior almacena en caché
este valor
```

## 8.5 Funciones como espacios de nombres

Las variables declaradas dentro de una función no son visibles fuera de ella. Por esta razón, a veces es útil definir una función simplemente para que actúe como un espacio de nombres temporal en el que se pueden definir variables sin abarrotar el espacio de nombres global.

Supongamos, por ejemplo, que tienes un trozo de código JavaScript que quieras utilizar en varios programas JavaScript diferentes (o, en el caso del JavaScript del cliente, en varias páginas web diferentes). Supongamos que este código, como la mayoría de los códigos, define variables para almacenar los resultados intermedios de su cálculo. El problema es que como este trozo de código se utilizará en muchos programas diferentes, no se sabe si las variables que crea entrarán en conflicto con las variables creadas por los programas que lo utilizan. La solución es

poner el trozo de código en una función y luego invocar la función. De esta manera, las variables que habrían sido globales se convierten en locales para la función:

```
function chunkNamespace() {  
    // El trozo de código va aquí  
    // Cualquier variable definida en el chunk es local a esta función // en lugar de  
    // abarrotar el espacio de nombres global. } chunkNamespace(); // ¡Pero no olvides  
    // invocar la función!
```

Este código define una sola variable global: el nombre de la función chunkNamespace. Si definir incluso una sola propiedad es demasiado, puedes definir e invocar una función anónima en una sola expresión:

```
(function() { // función chunkNamespace() reescrita como una expresión sin  
    // nombre. // El trozo de código va aquí }()); // Termina la función literal y la invoca  
    // ahora.
```

Esta técnica de definir e invocar una función en una sola expresión se utiliza con tanta frecuencia que se ha convertido en algo idiomático y se le ha dado el nombre de "expresión de función inmediatamente invocada". Observe el uso de paréntesis en el ejemplo de código anterior. El paréntesis abierto antes de la función es necesario porque, sin él, el intérprete de JavaScript intenta analizar la palabra clave de la función como una declaración de función. Con el paréntesis, el intérprete reconoce correctamente esto como una expresión de definición de función. El paréntesis inicial también ayuda a los lectores humanos a reconocer cuando una función está siendo definida para ser invocada inmediatamente en lugar de ser definida para su uso posterior.

Este uso de funciones como espacios de nombres se vuelve realmente útil cuando definimos una o más funciones dentro de la función del espacio de nombres usando variables dentro de ese espacio de nombres, pero luego las pasamos de vuelta como el valor de retorno de la función del espacio de nombres. Las funciones de este tipo se conocen como *cierres*, y son el tema de la siguiente sección.

## 8.6 Cierres

Como la mayoría de los lenguajes de programación modernos, JavaScript utiliza el ámbito *léxico*. Esto significa que las funciones se ejecutan utilizando el ámbito de la variable que estaba en vigor cuando se definieron, no el ámbito de la variable que está en vigor cuando se invocan. Para implementar el ámbito léxico, el estado interno de un objeto de función de JavaScript debe incluir no sólo el código de la función, sino también una referencia al ámbito en el que aparece la definición de la función. Esta combinación de un objeto de función y un ámbito (un conjunto de enlaces de variables) en el que se resuelven las variables de la función se denomina *cierre* en la literatura informática.

Técnicamente, todas las funciones de JavaScript son cierres, pero como la mayoría de las funciones son invocadas desde el mismo ámbito en el que fueron definidas, normalmente no importa que haya un cierre involucrado. Los cierres se vuelven interesantes cuando son invocados desde un ámbito diferente al que fueron definidos. Esto sucede más comúnmente cuando un objeto de función anidada es devuelto desde la función dentro de la cual fue definida. Hay una serie de poderosas técnicas de programación

que involucran este tipo de cierres de funciones anidadas, y su uso se ha vuelto relativamente común en la programación de JavaScript. Los cierres pueden parecer confusos cuando los encuentras por primera vez, pero es importante que los entiendas lo suficientemente bien como para utilizarlos cómodamente.

El primer paso para entender los cierres es revisar las reglas de alcance léxico para las funciones anidadas. Considere el siguiente código:

```
let scope = "ámbito global"; // Una variable global
function checkscope() { let
  scope = "ámbito local"; // Una variable local
  function f() { return scope; } //
  Devuelve el valor del ámbito aquí return f(); }
checkscope() //=> "ámbito local"
```

La función checkscope() declara una variable local y luego define e invoca una función que devuelve el valor de esa variable. Debería estar claro por qué la llamada a checkscope() devuelve "ámbito local". Ahora, cambiemos ligeramente el código. ¿Puedes decir qué devolverá este código?

```
let scope = "ámbito global"; // Una variable global
function checkscope() { let
  scope = "ámbito local"; // Una variable local
  function f() { return ámbito; } //
  Devuelve el valor del ámbito aquí return f(); } let s = checkscope(); // ¿Qué
devuelve esto?
```

En este código, un par de paréntesis se ha movido desde dentro de checkscope() a fuera de él. En lugar de invocar la función anidada y devolver su resultado, checkscope() ahora sólo devuelve el propio objeto de la función anidada. ¿Qué sucede cuando invocamos esa función anidada (con el segundo par de paréntesis en la última línea de código) fuera de la función en la que fue definida?

Recuerda la regla fundamental del ámbito léxico: Las funciones de JavaScript se ejecutan utilizando el ámbito en el que fueron definidas. La función anidada `f()` se definió en un ámbito en el que la variable `scope` estaba vinculada al valor "ámbito local". Esa vinculación sigue vigente cuando se ejecuta `f`, sin importar desde dónde se execute. Así que la última línea del ejemplo de código anterior devuelve "ámbito local", no "ámbito global". Esto, en pocas palabras, es la sorprendente y poderosa naturaleza de los cierres: capturan los enlaces de las variables locales (y los parámetros) de la función externa dentro de la cual están definidos.

En §8.4.1, definimos una función `uniqueInteger()` que utilizaba una propiedad de la propia función para llevar la cuenta del siguiente valor a devolver. Una de las deficiencias de este enfoque es que un código malicioso podría restablecer el contador o establecerlo en un valor no entero, haciendo que la función `uniqueInteger()` viole la parte "única" o "entera" de su contrato. Los cierres capturan las variables locales de una única invocación a la función y pueden utilizar esas variables como estado privado. Así es como podríamos reescribir la función `uniqueInteger()` utilizando una expresión de función inmediatamente invocada para definir un espacio de nombres y un cierre que utilice ese espacio de nombres para mantener su estado privado:

```
let uniqueInteger = (function() { // Definir e invocar let counter = 0; // Estado  
privado de la función siguiente  
    return function() { return counter++; }; }()); uniqueInteger()  
// => 0 uniqueInteger() // => 1
```

Para entender este código, hay que leerlo con atención. A primera vista, la primera línea de código parece que está asignando una función a la variable uniqueInteger. De hecho, el código está definiendo e invocando (como se desprende del paréntesis abierto en la primera línea) una función, por lo que es el valor de retorno de la función el que se asigna a uniqueInteger. Ahora, si estudiamos el cuerpo de la función, vemos que su valor de retorno es otra función. Es este objeto de función anidada el que se asigna a uniqueInteger. La función anidada tiene acceso a las variables de su ámbito y puede utilizar la variable contador definida en la función externa. Una vez que la función externa retorna, ningún otro código puede ver la variable contador: la función interna tiene acceso exclusivo a ella.

Las variables privadas como el contador no tienen por qué ser exclusivas de un único cierre: es perfectamente posible que dos o más funciones anidadas se definan dentro de la misma función externa y comparten el mismo ámbito.

Considere el siguiente código:

```
function counter() { let n = 0; return { count: function() {
  return n++; }, reset: function() { n = 0; } }; }

let c = counter(), d = counter(); // Crear dos contadores
c. count() // => 0
d. count() // => 0: cuentan independientemente
c. reset(); // los métodos reset() y count() comparten estado
c. count() // => 0: porque reiniciamos c
d. count() // => 1: d no se ha reiniciado
```

La función counter() devuelve un objeto "counter". Este objeto tiene dos métodos: count() devuelve el siguiente entero, y reset() reinicia el estado interno. Lo primero que hay que entender es que los dos métodos comparten el acceso a la variable privada n. Lo segundo que hay que entender es que cada invocación de counter() crea un nuevo ámbito -independiente de los ámbitos utilizados por las invocaciones anteriores- y una nueva variable privada dentro de ese ámbito. Así que si llamas a counter() dos veces, obtienes dos objetos contadores con diferentes variables privadas. Llamar a count() o a reset() en un objeto contador no tiene efecto en el otro.

Vale la pena señalar aquí que se puede combinar esta técnica de cierre con getters y setters de propiedades. La siguiente versión de la función counter() es una variación del código que apareció en [§6.10.6](#), pero utiliza cierres para el estado privado en lugar de depender de una propiedad regular del objeto:

```

function counter(n) { // El argumento de la función n es la variable privada return {
    // El método getter de la propiedad devuelve e incrementa el contador privado var.
    get count() { return n++; },
    // El setter de la propiedad no permite que el valor de n disminuya set count(m)
    { if (m > n) n = m;
        else throw Error("el recuento sólo puede establecerse en un valor
mayor");
    }
};

let c = contador(1000);
c. count // => 1000
c. count // => 1001
c. recuento = 2000;
c. count // => 2000
c. count = 2000; // !Error: count sólo puede ser fijado a un valor mayor

```

Observe que esta versión de la función counter() no declara una variable local, sino que sólo utiliza su parámetro n para mantener el estado privado compartido por los métodos de acceso a la propiedad. Esto permite a quien llama a counter() especificar el valor inicial de la variable privada.

El ejemplo 8-2 es una generalización del estado privado compartido a través de la técnica de cierres que hemos estado demostrando aquí. Este ejemplo define una función addPrivateProperty() que define una variable privada y dos funciones anidadas para obtener y establecer el valor de esa variable. Añade estas funciones anidadas como métodos del objeto que se especifica.

*Ejemplo 8-2. Métodos de acceso a propiedades privadas utilizando cierres*

---

```
// Esta función añade métodos de acceso a la propiedad para una propiedad con
// el nombre especificado al objeto o. Los métodos se denominan get<nombre>
// y set<nombre>. Si se suministra una función de predicado, el método setter // la utiliza
// para comprobar la validez de su argumento antes de almacenarlo. // Si el predicado
// devuelve false, el método setter lanza una excepción.
//
// Lo inusual de esta función es que el valor de la propiedad
// que es manipulado por los métodos getter y setter no se almacena en
// el objeto o. En cambio, el valor se almacena sólo en una variable local
// en esta función. Los métodos getter y setter también están definidos // localmente en
// esta función y por lo tanto tienen acceso a esta variable local.
// Esto significa que el valor es privado para los dos métodos de acceso, y que // no puede
// ser establecido o modificado excepto a través del método setter. function
addPrivateProperty(o, name, predicate) { let value; // Este es el valor de la propiedad

    // El método getter simplemente devuelve el valor. o[`get${name}`] = function() {
    return value; }

    // El método setter almacena el valor o lanza una excepción si // el
    // predicado rechaza el valor. o[`set${name}`] = function(v) { si (predicado &&
    ! predicado(v)) {
```

```

        throw new TypeError(`set${nombre}: valor inválido ${v}`);
    }
}

// El siguiente código demuestra el método addPrivateProperty(). let o = {};// Aquí
// hay un objeto vacío

// Añadir los métodos de acceso a la propiedad getName y setName()
// Asegúrese de que sólo se permiten valores de cadena addPrivateProperty(o, "Nombre",
x => typeof x === "string");

o.setName("Frank");// Establecer el valor de la propiedad
o.getName() //=> "Frank"
o.setName(0); // !TypeError: intento de establecer un valor de tipo incorrecto

```

Ya hemos visto varios ejemplos en los que dos cierres se definen en el mismo ámbito y comparten el acceso a la misma variable o variables privadas. Esta es una técnica importante, pero es igualmente importante reconocer cuando los cierres comparten inadvertidamente el acceso a una variable que no deberían compartir. Considere el siguiente código:

```

// Esta función devuelve una función que siempre devuelve v function constfunc(v) {
    return () => v;

// Crear una matriz de funciones constantes: let funcs = [];for(var i = 0; i <
10; i++) funcs[i] = constfunc(i);

// La función en el elemento 5 del array devuelve el valor 5.
funcs[5](); //=> 5

```

Cuando se trabaja con código como este que crea múltiples cierres utilizando un bucle, es un error común tratar de mover el bucle dentro de la función que define los cierres. Piensa en el siguiente código, por ejemplo:

```
// Devuelve un array de funciones que devuelven los valores 0-9
function constfuncs() { let funcs = []; for(var i = 0; i < 10; i++) { funcs[i] = () => i; } return funcs; }
```

```
let funcs = constfuncs(); funcs[5]() //=> 10; ¿Por qué no devuelve 5?
```

Este código crea 10 cierres y los almacena en un array. Los cierres están todos definidos dentro de la misma invocación de la función, por lo que comparten el acceso a la variable i. Cuando constfuncs() devuelve, el valor de la variable i es 10, y los 10 cierres comparten este valor. Por lo tanto, todas las funciones del array de funciones devuelven el mismo valor, que no es en absoluto lo que queríamos. Es importante recordar que el ámbito asociado a un cierre es "vivo". Las funciones anidadas no hacen copias privadas del ámbito ni hacen instantáneas estáticas de los enlaces de las variables. Fundamentalmente, el problema aquí es que las variables declaradas con var se definen a lo largo de la función. Nuestro bucle for declara la variable del bucle con var i, por lo que la variable i está definida en toda la función en lugar de tener un ámbito más limitado al cuerpo del bucle. El código demuestra una categoría común de errores en ES5 y anteriores, pero la introducción de variables de ámbito de bloque en ES6 resuelve el problema. Si sustituimos el var por un let o un const, el problema desaparece. Debido a que let y const tienen un ámbito de bloque, cada iteración del bucle define un ámbito que es independiente de los ámbitos de todas las demás iteraciones, y cada uno de estos ámbitos tiene su propio enlace independiente de i.

Otra cosa que hay que recordar al escribir cierres es que `this` es una palabra clave de JavaScript, no una variable. Como se ha dicho antes, las funciones de flecha heredan el valor `this` de la función que las contiene, pero las funciones definidas con la palabra clave `function` no lo hacen. Así que si estás escribiendo un cierre que necesita usar el valor `this` de su función contenedora, deberías usar una función de flecha, o llamar a `bind()`, en el cierre antes de devolverlo, o asignar el valor `this` externo a una variable que tu cierre heredará:

```
const self = this; // Hacer que el valor de this esté disponible para las funciones anidadas
```

## 8.7 Propiedades, métodos y constructores de funciones

Hemos visto que las funciones son valores en los programas de JavaScript. El operador `typeof` devuelve la cadena "función" cuando se aplica a una función, pero las funciones son realmente un tipo especializado de objeto de JavaScript. Como las funciones son objetos, pueden tener propiedades y métodos, como cualquier otro objeto. Incluso existe un constructor `Function()` para crear nuevos objetos de función. Las subsecciones siguientes documentan las propiedades `length`, `name` y `prototype`; los métodos `call()`, `apply()`, `bind()` y `toString()`; y el constructor `Function()`.

### 8.7.1 La propiedad de la longitud

La propiedad de longitud de sólo lectura de una función especifica la *aridad* de la función-el número de parámetros que declara en su

lista de parámetros, que suele ser el número de argumentos que la función espera. Si una función tiene un parámetro restante, ese parámetro no se cuenta a efectos de esta propiedad de longitud.

### 8.7.2 La propiedad nombre

La propiedad name de sólo lectura de una función especifica el nombre que se utilizó cuando se definió la función, si se definió con un nombre, o el nombre de la variable o propiedad a la que se asignó una expresión de función sin nombre cuando se creó por primera vez. Esta propiedad es principalmente útil cuando se escriben mensajes de depuración o de error.

### 8.7.3 La propiedad del prototipo

Todas las funciones, excepto las funciones de flecha, tienen una propiedad prototipo que se refiere a un objeto conocido como *objeto prototipo*. Cada función tiene un objeto prototipo diferente. Cuando una función se utiliza como constructor, el nuevo objeto creado hereda las propiedades del objeto prototipo. Los prototipos y la propiedad prototipo se trataron en [§6.2.3](#) y se volverán a tratar en el [capítulo 9](#).

### 8.7.4 Los métodos call() y apply()

call() y apply() permiten invocar indirectamente ([§8.2.4](#)) una función como si fuera un método de algún otro objeto. El primer argumento de call() y apply() es el objeto sobre el que se va a invocar la función; este argumento es el contexto de invocación y se convierte en el valor de la palabra clave this dentro del cuerpo de la función. Para invocar la función f() como un método del objeto o (sin pasar argumentos), puede utilizar call() o apply():

```
f. llamada(o);  
f. aplicar(o);
```

Cualquiera de estas líneas de código son similares a las siguientes (que suponen que o no tiene ya una propiedad llamada m):

```
o. m = f; // Hacer de f un método temporal de o.  
o. m(); // Invocarlo, sin pasar argumentos. delete o. m; // Eliminar el método temporal.
```

Recuerde que las funciones de flecha heredan el valor de this del contexto donde se definen. Esto no puede ser anulado con los métodos call() y apply(). Si llamas a cualquiera de esos métodos en una función de flecha, el primer argumento es efectivamente ignorado.

Todos los argumentos de call() después del primer argumento del contexto de invocación son los valores que se pasan a la función que se invoca (y estos argumentos no se ignoran para las funciones de flecha). Por ejemplo, para pasar dos números a la función f() e invocarla como si fuera un método del objeto o, se podría utilizar un código como el siguiente:

```
f. call(o, 1, 2);
```

El método apply() es como el método call(), excepto que los argumentos que se pasan a la función se especifican como una matriz:

```
f. aplicar(o, [1,2]);
```

Si una función está definida para aceptar un número arbitrario de argumentos, el método apply() permite invocar esa función sobre el contenido de un array de longitud arbitraria. En ES6 y

posteriores, podemos utilizar simplemente el operador spread, pero es posible que veas código ES5 que utiliza apply() en su lugar. Por ejemplo, para encontrar el mayor número de un array de números sin usar el operador spread, se puede usar el método apply() para pasar los elementos del array a la función Math.max():

```
let biggest = Math. max. apply(Math, arrayOfNumbers);
```

La función trace() definida a continuación es similar a la función timed() definida en §8.3.4, pero funciona para métodos en lugar de funciones. Utiliza el método apply() en lugar de un operador de propagación, y al hacerlo, es capaz de invocar el método envuelto con los mismos argumentos y el mismo valor de este que el método envolvente:

```
// Reemplazar el método llamado m del objeto o por una versión que registre // los
// mensajes antes y después de invocar el método original.
function trace(o, m) {
  let
    original = o[m]; // Recordar el método original en el cierre.

  o[m] = function(... args) { // Ahora define el nuevo método.
    console. log(new Date(), "Entering:", m); // Mensaje de registro.

    let result = original. apply(this, args); // Invocar original. console. log(new Date(),
    "Exiting:", m); // Mensaje de registro.

    return result; // Devuelve el resultado.
  };
}
```

## 8.7.5 El método bind()

El propósito principal de bind() es *vincular* una función a un objeto. Cuando se invoca el método bind() sobre una función f y se le pasa un objeto o, el método devuelve una nueva función. Al invocar la nueva función (como una función) se invoca la función original f como un método de o. Cualquier argumento que se pase a la nueva función se pasa a la función original. Por ejemplo:

```
function f(y) { return this.x + y; } // Esta función necesita ser enlazada let o = { x: 1 };
}; // Un objeto al que vamos a enlazar let g = f.bind(o); // Llamar a g(x) invoca a f()
en o.g(2) //=> 3 let p = { x: 10, g }; // Invoca a g() como un método de este objeto
p.g(2) //=> 3: g sigue ligado a o, no a p.
```

Las funciones de flecha heredan su valor del entorno en el que están definidas, y ese valor no puede anularse con bind(), por lo que si la función f() del código anterior estuviera definida como una función de flecha, la vinculación no funcionaría. Sin embargo, el caso de uso más común para llamar a bind() es hacer que las funciones que no son de flecha se comporten como funciones de flecha, por lo que esta limitación en la vinculación de funciones de flecha no es un problema en la práctica.

Sin embargo, el método bind() hace algo más que vincular una función a un objeto. También puede realizar una aplicación parcial: cualquier argumento que se pase a bind() después del primero se vincula junto con el valor de este.

Esta característica de aplicación parcial de bind() funciona con funciones de flecha. La aplicación parcial es una técnica común en la programación funcional y a veces se llama *currying*. Aquí hay algunos ejemplos del método bind() utilizado para la aplicación parcial:

```
let sum = (x,y) => x + y; // Devuelve la suma de 2 argumentos let succ = sum.
bind(null, 1); // Vincula el primer argumento a 1 succ(2) //=> 3: x está vinculado a
1, y pasamos 2 para el argumento y

function f(y,z) { return this.x + y + z; } let g = f.bind({x: 1}, 2); // Bind this and y g(3)
//=> 6: this.x está ligado a 1, y está ligado a 2 y z es 3
```

La propiedad name de la función devuelta por bind() es la propiedad name de la función sobre la que se llamó a bind(), prefijada con la palabra "bound".

### 8.7.6 El método `toString()`

Como todos los objetos de JavaScript, las funciones tienen un método `toString()`. La especificación ECMAScript requiere que este método devuelva una cadena que siga la sintaxis de la declaración de la función. En la práctica, la mayoría (pero no todas) de las implementaciones de este método `toString()` devuelven el código fuente completo de la función. Las funciones incorporadas suelen devolver una cadena que incluye algo como "[código nativo]" como cuerpo de la función.

### 8.7.7 El constructor `Function()`

Como las funciones son objetos, existe un constructor `Function()`

que pueden utilizarse para crear nuevas funciones:

```
const f = new Function("x", "y", "return x*y;");
```

Esta línea de código crea una nueva función que es más o menos equivalente a una función definida con la sintaxis conocida:

```
const f = function(x, y) { return x*y; };
```

El constructor de `Function()` espera cualquier número de argumentos de cadena. El último argumento es el texto del cuerpo de la función; puede contener declaraciones JavaScript arbitrarias, separadas entre sí por punto y coma. Todos los demás argumentos del constructor son cadenas que especifican los nombres de los

parámetros de la función. Si está definiendo una función que no toma argumentos, simplemente pasaría una sola cadena -el cuerpo de la función- al constructor.

Observe que al constructor Function() no se le pasa ningún argumento que especifique un nombre para la función que crea. Al igual que los literales de función, el constructor Function() crea funciones anónimas.

Hay algunos puntos que es importante entender sobre el constructor Function():

- El constructor Function() permite crear y compilar dinámicamente funciones de JavaScript en tiempo de ejecución.
- El constructor de Function() analiza el cuerpo de la función y crea un nuevo objeto de función cada vez que es llamado. Si la llamada al constructor aparece dentro de un bucle o dentro de una función llamada con frecuencia, este proceso puede ser ineficiente. En cambio, las funciones anidadas y las expresiones de función que aparecen dentro de bucles no se recompilan cada vez que se encuentran.
- Un último punto muy importante sobre el constructor Function() es que las funciones que crea no utilizan el ámbito léxico; en su lugar, siempre se compilan como si fueran funciones de nivel superior, como demuestra el siguiente código:

```
let scope = "global"; function constructFunction() { let
scope = "local"; return new Function("return scope"); //  
// No captura el ámbito local!
}  
// Esta línea devuelve "global" porque la función  
// devuelta por el constructor // de Function() no utiliza el  
// ámbito local.  
constructFunction()() //=> "global"
```

El constructor `Function()` es una versión global de `eval()` (ver [§4.12.2](#)) que define nuevas variables y funciones en su propio ámbito privado. Probablemente nunca necesitará utilizar este constructor en su código.

## 8.8 Programación funcional

JavaScript no es un lenguaje de programación funcional como Lisp o Haskell, pero el hecho de que JavaScript pueda manipular funciones como objetos significa que podemos utilizar técnicas de programación funcional en JavaScript. Los métodos de arrays como `map()` y `reduce()` se prestan especialmente bien a un estilo de programación funcional. Las secciones siguientes muestran técnicas de programación funcional en JavaScript. Están pensadas como una exploración de la potencia de las funciones de JavaScript, no como una prescripción para un buen estilo de programación.

### 8.8.1 Procesamiento de matrices con funciones

Supongamos que tenemos una matriz de números y queremos calcular la media y la desviación estándar de esos valores.

Podríamos hacerlo en un estilo no funcional como este:

```
let data = [1,1,3,5,5]; // Esta es nuestra matriz de números

// La media es la suma de los elementos dividida por el número de elementos let
total = 0; for(let i = 0; i < data.length; i++) total += data[i]; let mean = total / data.length; // mean == 3; La media de nuestros datos es 3

// Para calcular la desviación estándar, primero sumamos los cuadrados de // la
// desviación de cada elemento respecto a la media. total = 0; for(let i = 0; i < data.length; i++)
{ let deviation = data[i] - mean; total += deviation * deviation; } let
stddev = Math.sqrt(total / (data.length - 1)); // stddev == 2
```

Podemos realizar estos mismos cálculos en un estilo funcional conciso utilizando los métodos de matriz `map()` y `reduce()` de esta manera (ver §7.8.1 para revisar estos métodos):

```
// Primero, define dos funciones simples const sum =
(x, y) => x + y; const square = x => x * x;

// A continuación, utilice esas funciones con los métodos Array para calcular la
// media y el stddev let data = [1,1,3,5,5]; let mean = data.reduce(sum) / data.length;
// mean == 3 let deviations = data.map(x => x - mean); let stddev = Math.
sqrt(deviations.map(square).reduce(sum) / (data.length -
1)); stddev // == 2
```

Esta nueva versión del código tiene un aspecto bastante diferente al de la primera, pero sigue invocando métodos sobre objetos, por lo que conserva algunas convenciones orientadas a objetos. Vamos a escribir versiones funcionales de los métodos `map()` y `reduce()`:

```
const map = function(a, ... args) { return a.map(... args); };
const reduce = function(a, ... args) { return
a.reduce(... args); };
```

Con estas funciones map() y reduce() definidas, nuestro código para calcular la media y la desviación estándar tiene ahora este aspecto:

```
const suma = (x,y) => x+y; const cuadrado  
= x => x*x;  
  
let data = [1,1,3,5,5]; let mean = reduce(data, sum)/data.length; let  
deviations = map(data, x => x-mean); let stddev = Math.  
sqrt(reduce(map(deviations, square), sum)/(data.length-1)); stddev //=> 2
```

## 8.8.2 Funciones de orden superior

Una *función de orden superior* es una función que opera sobre funciones, tomando una o más funciones como argumentos y devolviendo una nueva función. He aquí un ejemplo:

```
// Esta función de orden superior devuelve una nueva función que pasa su  
// argumentos a f y devuelve la negación lógica del valor de retorno de f; function  
not(f) { return function(... args) { // Devuelve una nueva función let result = f.  
apply(this, args); // que llama a f return ! result; // y niega su resultado. }; }  
  
const even = x => x % 2 === 0; // Una función para determinar si un número es par  
const odd = not(even); // Una nueva función que hace lo contrario [1,1,3,5,5].  
every(odd) // => true: cada elemento del array es impar
```

Esta función not() es una función de orden superior porque toma un argumento de función y devuelve una nueva función. Como otro ejemplo, considere la función mapper() que sigue. Toma un argumento de función y devuelve una nueva función que mapea un array a otro usando esa función. Esta función utiliza la función map() definida anteriormente, y es importante que entiendas en qué se diferencian ambas funciones:

```
// Devuelve una función que espera un argumento de matriz y aplica f a // cada  
// elemento, devolviendo la matriz de valores de retorno. // Contrasta esto con la  
// función map() de antes.
```

```

función mapper(f) {
    return a => map(a, f); }

const increment = x => x+1; const incrementAll =
mapper(increment); incrementAll([1,2,3]) // => [2,3,4]

```

Aquí hay otro ejemplo más general que toma dos funciones, f y g, y devuelve una nueva función que calcula f(g()):

```

// Devuelve una nueva función que calcula f(g(...)).
// La función devuelta h pasa todos sus argumentos a g, luego pasa // el valor de
// retorno de g a f, y luego devuelve el valor de retorno de f. // Tanto f como g son
// invocadas con el mismo valor de este con el que fue invocada h.
function compose(f, g) { return function(...
args) {
    // Usamos call para f porque estamos pasando un único valor y // apply para g
    // porque estamos pasando un array de valores. return f. call(this, g. apply(this, args));
}; }

const sum = (x,y) => x+y; const square = x => x*x; compose(square, sum)(2,3) // =>
25; el cuadrado de la suma

```

Las funciones partial() y memoize() definidas en las secciones siguientes son dos funciones de orden superior más importantes.

### 8.8.3 Aplicación parcial de funciones

El método bind() de una función f (véase §8.7.5) devuelve una nueva función que invoca a f en un contexto y con un conjunto de argumentos especificados. Decimos que vincula la función a un objeto y aplica parcialmente los argumentos. El método bind() aplica parcialmente los argumentos por la izquierda, es decir, los argumentos que se pasan a bind() se colocan al principio de la lista

de argumentos que se pasa a la función original. Pero también es posible aplicar parcialmente los argumentos por la derecha:

```
// Los argumentos de esta función se pasan por la izquierda function partialLeft(f, ...
outerArgs) { return function(... innerArgs) { // Devuelve esta función let args = [...,
outerArgs, ... innerArgs]; // Construye la lista de argumentos return f. apply(this,
args); // Luego invoca f con ella };
}

// Los argumentos de esta función se pasan por la derecha function partialRight(f, ...
outerArgs) { return function(... innerArgs) { // Devuelve esta función let args = [...,
innerArgs, ... outerArgs]; // Construye la lista de argumentos return f. apply(this,
args); // Luego invoca f con ella };
}

// Los argumentos de esta función sirven como plantilla.
Valores indefinidos
// en la lista de argumentos se rellenan con valores del conjunto interno.
function partial(f, ... outerArgs) { return function(... innerArgs) { let args = [...,
outerArgs]; // copia local de la plantilla de args externos let innerIndex=0; // qué arg
interno es el siguiente
    // Recorre los argumentos, rellenando los valores no definidos de los
    argumentos internos for(let i = 0; i < args.length; i++) {
        if (args[i] === undefined) args[i] = innerArgs[innerIndex++];
    }      // Ahora añade los argumentos internos restantes args.push(
    innerArgs.slice(innerIndex)); return f. apply(this, args); };
}

// Aquí tenemos una función con tres argumentos const f = function(x,y,z) { return x * (y
- z); }; // Observa cómo difieren estas tres aplicaciones parciales partialLeft(f, 2)(3,4) //
=> -2: Bind first argument: 2 * (3 - 4) partialRight(f, 2)(3,4) // => 6: Bind last argument: 3
* (4 - 2) partial(f, undefined, 2)(3,4) // => -6: Bind middle argument: 3 * (2 - 4)
```

Estas funciones de aplicación parcial nos permiten definir fácilmente funciones interesantes a partir de funciones que ya tenemos definidas. He aquí algunos ejemplos:

```
const increment = partialLeft(sum, 1); const cuberoot =  
partialRight(Math. pow, 1/3); cuberoot(increment(26)) //  
=> 3
```

La aplicación parcial se vuelve aún más interesante cuando la combinamos con otras funciones de orden superior. Aquí, por ejemplo, hay una forma de definir la función not() anterior que acabamos de mostrar utilizando la composición y la aplicación parcial:

```
const not = partialLeft(compose, x => ! x); const even = x  
=> x % 2 === 0; const odd = not(even); const isNumber  
= not(isNaN); odd(3) && isNumber(2) // => true
```

También podemos utilizar la composición y la aplicación parcial para rehacer nuestros cálculos de la media y la desviación estándar en estilo funcional extremo:

```

// Las funciones sum() y square() están definidas más arriba. Aquí hay algunas más:
const product = (x,y) => x*y; const neg = partial(product, -1); const sqrt =
partial(Math. pow, undefined, . 5); const reciprocal = partial(Math. pow,
undefined, neg(1));

// Ahora calcula la media y la desviación estándar.
let data = [1,1,3,5,5]; // Nuestros datos let mean = product(reduce(data, sum),
reciprocal(data. length)); let stddev = sqrt(product(reduce(map(data,
compose(square, partial(sum, neg(mean))))), sum), reciprocal(sum(data.
length,neg(1)))));

[media, stddev] //=> [3, 2]

```

Observe que este código para calcular la media y la desviación estándar es enteramente invocaciones de funciones; no hay operadores involucrados, y el número de paréntesis ha crecido tanto que este JavaScript está empezando a parecer código Lisp. De nuevo, este no es un estilo que yo defienda para la programación en JavaScript, pero es un ejercicio interesante para ver lo profundamente funcional que puede ser el código JavaScript.

#### 8.8.4 Memoización

En §8.4.1, definimos una función factorial que almacena en caché sus resultados previamente calculados. En programación funcional, este tipo de almacenamiento en caché se llama *memoización*. El código que sigue muestra una función de orden superior, memoize(), que acepta una función como su argumento y devuelve una versión memoizada de la función:

```

// Devuelve una versión memoizada de f.
// Sólo funciona si los argumentos de f tienen todos representaciones de cadena
// distintas.
function memoize(f) { const cache = new Map(); // Caché de valores almacenados
en el cierre.

return function(... args) {
    // Crear una versión de cadena de los argumentos para usar como clave de caché.
    let key = args.length + args.join("+"); if (cache.has(key)) {
        return cache.get(key); } else { let result = f. apply(this, args); cache.
set(key, result); return result; }
};

}

```

La función memoize() crea un nuevo objeto para utilizarlo como caché y asigna este objeto a una variable local para que sea privado para (en el cierre de) la función devuelta. La función devuelta convierte su matriz de argumentos en una cadena y utiliza esa cadena como nombre de propiedad para el objeto de la caché. Si existe un valor en la caché, lo devuelve directamente. En caso contrario, llama a la función especificada para calcular el valor de estos argumentos, almacena ese valor en la caché y lo devuelve. Así es como podríamos utilizar memoize():

```

// Devuelve el Máximo Común Divisor de dos enteros utilizando el algoritmo //
euclidiano:

```

```

http://en.wikipedia.org/wiki/Euclidean_algorithm
function gcd(a,b) { // Se ha omitido la comprobación de tipo para a y b
  si (a < b) { // Asegura que a >= b cuando empezamos
    [a, b] = [b, a]; // Desestructuración de la asignación para intercambiar variables
  }
  while(b !== 0) { // Este es el algoritmo de Euclides para el GCD
    [a, b] = [b, a%b];
  }
  return a;
}

const gcdmemo = memoize(gcd);
gcmemo(85, 187) //
=> 17

// Ten en cuenta que cuando escribimos una función recursiva que vamos a memoizar,
// normalmente queremos recurrir a la versión memoizada, no a la original.
const factorial = memoize(function(n) {
  return (n <= 1) ? 1 : n * factorial(n-1);
});
factorial(5) // => 120: también memoriza valores para 4, 3, 2 y 1.

```

## 8.9 Resumen

Algunos puntos clave que hay que recordar sobre este capítulo son los siguientes:

- Puedes definir funciones con la palabra clave `function` y con la sintaxis de flechas ES6 `=>`.
- Puede invocar funciones, que pueden ser utilizadas como métodos y constructores.
- Algunas funciones de ES6 permiten definir valores por defecto para

parámetros opcionales de la función, para reunir múltiples argumentos en un array utilizando un parámetro de reposo, y para desestructurar argumentos de objetos y arrays en parámetros de función.

- Puedes utilizar el operador de propagación ... para pasar los elementos de un array u otro objeto iterable como argumentos en una invocación de función.
- Una función definida dentro de una función envolvente y devuelta por ella conserva el acceso a su ámbito léxico y, por tanto, puede leer y escribir las variables definidas dentro de la función externa. Las funciones utilizadas de este modo se denominan *cierres*, y es una técnica que merece la pena comprender.
- Las funciones son objetos que pueden ser manipulados por JavaScript, y esto permite un estilo de programación funcional.

---

El término fue acuñado por Martin Fowler. Véase

<sup>1</sup><http://martinfowler.com/dslCatalog/methodChaining.html>.

Si está familiarizado con Python, tenga en cuenta que esto es diferente a Python, en el que cada <sup>2</sup>invocación comparte el mismo valor por defecto.

Esto puede no parecer un punto particularmente interesante a menos que esté familiarizado con más <sup>3</sup>lenguajes estáticos, en los que las funciones forman parte de un programa pero no pueden ser manipuladas por el programa.



# Capítulo 9. Clases

---

Los objetos de JavaScript se trataron en [el capítulo 6](#). En ese capítulo se trató cada objeto como un conjunto único de propiedades, diferente de cualquier otro objeto. Sin embargo, a menudo es útil definir una *clase* de objetos que comparten ciertas propiedades. Los miembros, o *instancias*, de la clase tienen sus propias propiedades para mantener o definir su estado, pero también tienen métodos que definen su comportamiento. Estos métodos son definidos por la clase y compartidos por todas las instancias. Imagina una clase llamada `Complejo` que representa y realiza aritmética sobre números complejos, por ejemplo. Una instancia de `Complex` tendría propiedades para mantener las partes real e imaginaria (el estado) del número complejo. Y la clase `Complex` definiría métodos para realizar la suma y la multiplicación (el comportamiento) de esos números.

En JavaScript, las clases utilizan la herencia basada en prototipos: si dos objetos heredan propiedades (generalmente propiedades con valor de función, o métodos) del mismo prototipo, entonces decimos que esos objetos son instancias de la misma clase. Así, en pocas palabras, es como funcionan las clases de JavaScript. Los prototipos y la herencia de JavaScript se trataron en [§6.2.3](#) y [§6.3.2](#), y necesitarás estar familiarizado con el material de esas secciones para entender este capítulo. Este capítulo cubre los prototipos en [§9.1](#).

Si dos objetos heredan del mismo prototipo, esto significa típicamente (pero no necesariamente) que fueron creados e inicializados por la misma función constructora o función de fábrica. Los constructores han sido cubiertos en [§4.6](#), [§6.2.2](#) y [§8.2.3](#), y este capítulo tiene más en [§9.2](#).

JavaScript siempre ha permitido la definición de clases. ES6 introduce una nueva sintaxis (incluyendo una palabra clave `class`) que hace aún más fácil la creación de clases. Estas nuevas clases de JavaScript funcionan de la misma manera que las clases de estilo antiguo, y este capítulo comienza explicando la forma antigua de crear clases porque eso demuestra más claramente lo que sucede detrás de las escenas para que las clases funcionen. Una vez que hayamos explicado esos fundamentos, cambiaremos y empezaremos a usar la nueva y simplificada sintaxis de definición de clases.

Si está familiarizado con los lenguajes de programación orientados a objetos fuertemente tipados, como Java o C++, se dará cuenta de que las clases de JavaScript son bastante diferentes de las clases de esos lenguajes. Hay algunas similitudes sintácticas, y se pueden emular muchas características de las clases "clásicas" en JavaScript, pero es mejor entender por adelantado que las clases de JavaScript y el mecanismo de herencia basado en prototipos son sustancialmente diferentes de las clases y el mecanismo de herencia basado en clases de Java y lenguajes similares.

## 9.1 Clases y prototipos

En JavaScript, una clase es un conjunto de objetos que heredan propiedades del mismo objeto prototipo. El objeto prototipo, por lo tanto, es la característica central de una clase. En [el capítulo 6](#) se trató la función `Object.create()` que devuelve un objeto recién creado que hereda de un objeto prototipo especificado. Si definimos un objeto prototipo y luego usamos `Object.create()` para crear objetos que heredan de él, hemos definido una clase de JavaScript. Normalmente, las instancias de una clase requieren una inicialización adicional, y es común definir una función que cree e inicialice el nuevo objeto. [El ejemplo 9-1](#) demuestra esto: define un objeto prototipo para una clase que representa un rango de valores y también define una función de *fábrica* que crea e inicializa una nueva instancia de la clase.

*Ejemplo 9-1. Una clase simple de JavaScript*

---

```
// Esta es una función de fábrica que devuelve un nuevo objeto range. function
range(from, to) {
    // Utilice Object.create() para crear un objeto que herede del
    // objeto prototipo definido a continuación. El objeto prototipo se almacena como
    // una propiedad de esta función, y define los métodos compartidos
    // (comportamiento) para todos los objetos range. let r = Object.create(range.
    methods);

    // Almacena los puntos inicial y final (estado) de este nuevo objeto de rango. // Estas
    // son propiedades no heredadas que son únicas para este objeto.
    r.de = de;
    r.a = a;

    // Finalmente devuelve el nuevo objeto return r; }

// Este objeto prototipo define los métodos heredados por todos los objetos de rango.
rango.métodos = {
    // Devuelve true si x está en el rango, false en caso contrario
    // Este método funciona tanto para rangos textuales y de fechas como numéricos.
    includes(x) { return this.from <= x && x <= this.to; },

    // Una función generadora que hace que las instancias de la clase sean iterables. //
    // Tenga en cuenta que sólo funciona para rangos numéricos.
```

```

*[Symbol.iterator]() { for(let x = Math.ceil(this.from); x <= this.to; x++) yield x; }

// Devuelve una representación de cadena del rango toString() { devuelve "(" + this.
from + "... " + this.to + ")"; }

};

// Aquí hay ejemplos de uso de un objeto de rango.
let r = range(1,3); // Crear un objeto range
r.includes(2) //=> true: 2 está en el rango
r.toString() //=> "(1...3)"
[... r] //=> [1, 2, 3]; convertir en un array mediante un iterador

```

Hay algunas cosas que vale la pena señalar en el código del Ejemplo 9-1:

- Este código define una función de fábrica range() para crear nuevos objetos Range.
- Utiliza la propiedad methods de esta función range() como un lugar conveniente para almacenar el objeto prototipo que define la clase. No hay nada especial o idiomático en poner el objeto prototipo aquí.
- La función range() define las propiedades from y to en cada objeto Range. Estas son las propiedades no compartidas y no heredadas que definen el estado único de cada objeto Range individual.
- El objeto range.methods utiliza la sintaxis abreviada de ES6 para definir métodos, por lo que no se ve la palabra clave function en ninguna parte. (Ver [§6.10.5](#) para revisar la sintaxis literal de métodos del objeto).
- Uno de los métodos del prototipo tiene el nombre calculado ([§6.10.2](#)) Symbol.iterator, lo que significa que es

que define un iterador para los objetos Range. El nombre de este método lleva el prefijo `*`, que indica que es una función generadora en lugar de una función normal. Los iteradores y generadores se tratan en detalle en [el capítulo 12](#). Por ahora, el resultado es que las instancias de esta clase Range pueden utilizarse con el bucle `for/of` y con el operador de propagación ....

- Los métodos compartidos y heredados definidos en `range.methods` utilizan las propiedades `from` y `to` que fueron inicializadas en la función de fábrica `range()`. Para referirse a ellos, utilizan la palabra clave `this` para referirse al objeto a través del cual fueron invocados. Este uso de `this` es una característica fundamental de los métodos de cualquier clase.

## 9.2 Clases y Constructores

[El ejemplo 9-1](#) demuestra una forma sencilla de definir una clase JavaScript. Sin embargo, no es la forma idiomática de hacerlo, porque no define un *constructor*. Un constructor es una función diseñada para la inicialización de objetos recién creados. Los constructores se invocan utilizando la palabra clave `new`, como se describe en [§8.2.3](#). Las invocaciones a constructores usando `new` crean automáticamente el nuevo objeto, por lo que el constructor mismo sólo necesita inicializar el estado de ese nuevo objeto. La característica crítica de las invocaciones al constructor es que la propiedad `prototype` del constructor se utiliza como prototipo del nuevo objeto. En [§6.2.3](#) se introdujeron los prototipos y se hizo hincapié en que, aunque casi todos los objetos tienen un prototipo, sólo unos pocos objetos tienen una propiedad prototipo. Finalmente, podemos aclarar esto: son los objetos función los que tienen una propiedad prototipo. Esto significa que

todos los objetos creados con la misma función constructora heredan del mismo objeto y, por tanto, son miembros de la misma clase. El Ejemplo 9-2 muestra cómo podríamos modificar la clase Range del Ejemplo 9-1 para utilizar una función constructora en lugar de una función de fábrica. El Ejemplo 9-2 demuestra la forma idiomática de crear una clase en versiones de JavaScript que no soportan la palabra clave class de ES6. Aunque la clase está bien soportada ahora, todavía hay mucho código JavaScript antiguo que define clases como esta, y deberías estar familiarizado con el lenguaje para poder leer el código antiguo y para que entiendas lo que está pasando "bajo el capó" cuando usas la palabra clave class.

*Ejemplo 9-2. Una clase Range que utiliza un constructor*

---

```

// Esta es una función constructora que inicializa nuevos objetos Range. // Tenga en cuenta que no crea ni devuelve el objeto. Sólo lo inicializa.
función Rango(desde, hasta) {
    // Almacena los puntos inicial y final (estado) de este nuevo objeto de rango. // Estas son propiedades no heredadas que son únicas para este objeto.
    this. from = from; this. to =
    to; }

// Todos los objetos Range heredan de este objeto.
// Tenga en cuenta que el nombre de la propiedad debe ser "prototipo" para que esto funcione. Rango. prototipo = {
    // Devuelve true si x está en el rango, false en caso contrario
    // Este método funciona para rangos textuales y de fechas, así como numéricos.
    includes: function(x) { return this. from <= x && x <= this. to; },

    // Una función generadora que hace que las instancias de la clase sean iterables. // Tenga en cuenta que sólo funciona para rangos numéricos.
    [Símbolo. iterador]: function*() {
        for(let x = Math. ceil(this. from); x <= this. to; x++) yield x; }

    // Devuelve una representación de cadena del rango toString: function() { return "("
    + this. from + "... " + this. to + ")"; } };

// Aquí hay ejemplos de uso de esta nueva clase Range let r = new Range(1,3); // Crear un objeto Range; nótese el uso de new
r. includes(2) // => true: 2 está en el rango
r. toString() // => "(1...3)"
[... r] // => [1, 2, 3]; convertir en un array mediante un iterador

```

Vale la pena comparar los Ejemplos [9-1](#) y [9-2](#) con bastante cuidado y observar las diferencias entre estas dos técnicas para definir

clases. En primer lugar, observe que hemos renombrado la función de fábrica range() a Range() cuando la convertimos en un constructor. Esta es una convención de codificación muy común: las funciones constructoras definen, en cierto sentido, clases, y las clases tienen nombres que (por convención) comienzan con letras mayúsculas. Las funciones y métodos regulares tienen nombres que comienzan con letras minúsculas.

A continuación, observe que el constructor Range() es invocado (al final del ejemplo) con la palabra clave new mientras que la función de fábrica range() fue invocada sin ella. El [Ejemplo 9-1](#) utiliza la invocación regular de la función ([§8.2.1](#)) para crear el nuevo objeto, y el [Ejemplo 9-2](#) utiliza la invocación del constructor ([§8.2.3](#)). Debido a que el constructor Range() es invocado con new, no tiene que llamar a Object.create() ni realizar ninguna acción para crear un nuevo objeto. El nuevo objeto se crea automáticamente antes de llamar al constructor, y es accesible como el valor this. El constructor Range() simplemente tiene que inicializar this.

Los constructores ni siquiera tienen que devolver el objeto recién creado. La invocación de un constructor crea automáticamente un nuevo objeto, invoca el constructor como un método de ese objeto y devuelve el nuevo objeto. El hecho de que la invocación de un constructor sea tan diferente de la invocación de una función normal es otra de las razones por las que damos a los constructores nombres que empiezan con mayúsculas. Los constructores están escritos para ser invocados como constructores, con la palabra clave new, y normalmente no funcionarán correctamente si son invocados como funciones regulares. Una convención de nomenclatura que mantiene a las

funciones constructoras distintas de las funciones regulares ayuda a los programadores a saber cuándo usar new.

### Y NEW.TARGET

En un cuerpo de función, se puede saber si la función ha sido invocada como un constructor con la etiqueta expresión especial `nuevo.objeto`. Si el valor de esa expresión está definido, entonces se sabe que el fue invocada como un constructor, con la función `nuev` palabra clave. Cuando hablamos de subclases, en veremos que `nuevo.objeto` siempre es una referencia al constructor en el que se utiliza: también puede referirse a a la función constructora de una subclase.

Si `nuevo.objeto` indefinido entonces la función contenedora fue invocada como una función, sin el `nuev` palabra clave. Los diversos constructores de errores de JavaScript pueden ser imitados sin emular esta característica en tus propios constructores, puedes escribirlos así:

```
función C() {  
    si (!nuev.objeto) devolveruev C();  
    // el código de inicialización va aquí  
}
```

Esta técnica sólo funciona para los constructores definidos de esta forma antigua. Las clases creadas con el método `clase` no permiten que sus constructores sean invocados sin `nuev`

Otra diferencia crítica entre los Ejemplos 9-1 y 9-2 es la forma en que se nombra el objeto prototipo. En el primer ejemplo, el prototipo era `range.methods`. Este era un nombre conveniente y descriptivo, pero arbitrario. En el segundo ejemplo, el prototipo es `Range.prototype`, y este nombre es obligatorio. Una invocación del constructor `Range()` utiliza automáticamente `Range.prototype` como prototipo del nuevo objeto `Range`.

Finalmente, observe también las cosas que no cambian entre los Ejemplos 9-1 y 9-2: los métodos de rango se definen e invocan de la misma manera para ambas clases. Debido a que el Ejemplo 9-2 demuestra la forma idiomática de crear clases en versiones de JavaScript anteriores a ES6, no utiliza la sintaxis abreviada de métodos de ES6 en el objeto prototipo y deletrea explícitamente

los métodos con la palabra clave `function`. Pero puede ver que la implementación de los métodos es la misma en ambos ejemplos.

Es importante señalar que ninguno de los dos ejemplos de rango utiliza funciones de flecha al definir constructores o métodos.

Recuerde de §8.1.3 que las funciones definidas de esta manera no tienen una propiedad prototipo y por lo tanto no pueden ser utilizadas como constructores. Además, las funciones de flecha heredan la palabra clave `this` del contexto en el que se definen en lugar de establecerla en función del objeto a través del cual se invocan, y esto las hace inútiles para los métodos porque la característica que define a los métodos es que utilizan `this` para referirse a la instancia sobre la que fueron invocados.

Afortunadamente, la nueva sintaxis de la clase ES6 no permite la opción de definir métodos con funciones de flecha, por lo que esto no es un error que se pueda cometer accidentalmente al usar esa sintaxis. Cubriremos la palabra clave de la clase ES6 pronto, pero primero, hay más detalles que cubrir sobre los constructores.

### 9.2.1 Constructores, identidad de clase y `instanceof`

Como hemos visto, el objeto prototipo es fundamental para la identidad de una clase: dos objetos son instancias de la misma clase si y sólo si heredan del mismo objeto prototipo. La función constructora que inicializa el estado de un nuevo objeto no es fundamental: dos funciones constructoras pueden tener propiedades prototipo que apunten al mismo objeto prototipo. Entonces, ambos constructores pueden utilizarse para crear instancias de la misma clase.

Aunque los constructores no son tan fundamentales como los prototipos, el constructor sirve como cara pública de una clase. Lo más evidente es que el nombre de la función del constructor suele adoptarse como nombre de la clase. Decimos, por ejemplo, que el constructor Range() crea objetos Range. Sin embargo, los constructores se utilizan como el operando derecho del operador instanceof cuando se comprueba la pertenencia de los objetos a una clase. Si tenemos un objeto r y queremos saber si es un objeto Range, podemos escribir

```
r instanceof Range // => true: r hereda de Range.prototype
```

El operador instanceof se describió en §4.9.4. El operando izquierdo debe ser el objeto que se está probando, y el operando derecho debe ser una función constructora que nombra una clase. La expresión o instanceof C se evalúa como verdadera si o hereda de C.prototype. La herencia no tiene por qué ser directa: si o hereda de un objeto que hereda de un objeto que hereda de C.prototype, la expresión seguirá siendo verdadera.

Técnicamente hablando, en el ejemplo de código anterior, el operador instanceof no está comprobando si r fue realmente inicializado por el constructor de Range. En su lugar, está comprobando si r hereda de Range.prototype. Si definimos una función

Strange() y establecer su prototipo para que sea el mismo que Range.prototype, entonces los objetos creados con new Strange() contarán como objetos Range en lo que respecta a instanceof (sin embargo, no funcionarán realmente como objetos Range, porque sus propiedades from y to no han sido inicializadas):

```
function Strange() {} Strange.prototype = Range.prototype;  
new Strange() instanceof Range // => true
```

Aunque `instanceof` no puede verificar realmente el uso de un constructor, sigue utilizando una función constructora como su lado derecho porque los constructores son la identidad pública de una clase.

Si quiere probar la cadena de prototipos de un objeto para un prototipo específico y no quiere utilizar la función constructora como intermediaria, puede utilizar el método `isPrototypeOf()`. En [el Ejemplo 9-1](#), por ejemplo, definimos una clase sin función constructora, por lo que no hay forma de utilizar `instanceof` con esa clase. En su lugar, sin embargo, podríamos comprobar si un objeto `r` era un miembro de esa clase sin constructor con este código:

```
range.methods.isPrototypeOf(r); // range.methods es el objeto prototipo.
```

### 9.2.2 La propiedad del constructor

En [el Ejemplo 9-2](#), establecimos `Range.prototype` en un nuevo objeto que contenía los métodos de nuestra clase. Aunque era conveniente expresar esos métodos como propiedades de un único objeto literal, no era realmente necesario crear un nuevo objeto. Cualquier función regular de JavaScript (excluyendo las funciones de flecha, las funciones generadoras y las funciones asíncronas) puede utilizarse como constructor, y las invocaciones de constructores necesitan una propiedad prototipo. Por lo tanto, toda función regular de JavaScript tiene automáticamente una propiedad prototipo. El valor de esta propiedad es un objeto que

tiene una única propiedad constructora no enumerable. El valor de la propiedad del constructor es el objeto de la función:

```
let F = function() {} // Este es un objeto función. let p = F.prototype; // Este es el objeto prototipo asociado a F.  
let c = p.constructor; // Esta es la función asociada al prototipo.  
c === F // => true: F.prototype.constructor  
==== F para cualquier F
```

La existencia de este objeto prototipo predefinido con su propiedad constructora significa que los objetos suelen heredar una propiedad constructora que hace referencia a su constructor. Dado que los constructores sirven como la identidad pública de una clase, esta propiedad constructora da la clase de un objeto:

```
let o = new F(); // Crear un objeto o de la clase F  
o.constructor === F // => true: la propiedad constructor especifica la clase
```

La Figura 9-1 ilustra esta relación entre la función del constructor, su objeto prototipo, la referencia posterior del prototipo al constructor y las instancias creadas con el constructor.

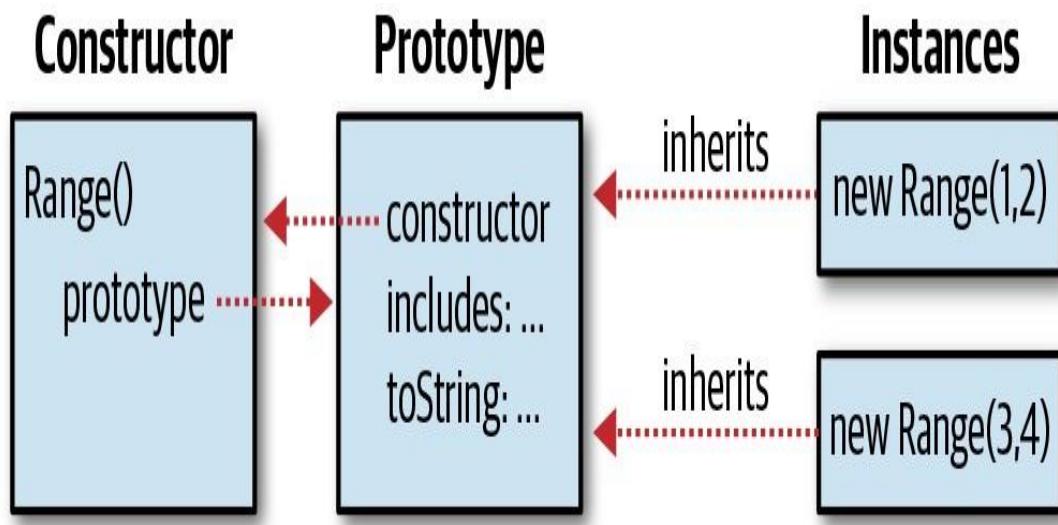


Figura 9-1. Una función constructora, su prototipo y las instancias

Observe que la Figura 9-1 utiliza nuestro constructor Range() como ejemplo. De hecho, sin embargo, la clase Range definida en el Ejemplo 9-2 sobrescribe el objeto predefinido Range.prototype con un objeto propio. Y el nuevo objeto prototipo que define no tiene una propiedad constructora. Así que las instancias de la clase Range, tal y como están definidas, no tienen una propiedad constructora. Podemos remediar este problema añadiendo explícitamente un constructor al prototipo:

```
Range.prototype = { constructor: Range, // Establece explícitamente la referencia  
posterior del constructor  
  
/* las definiciones de los métodos van aquí */  
};
```

Otra técnica común que es probable que vea en las El código JavaScript consiste en utilizar el objeto prototipo predefinido con su propiedad constructora y añadirle métodos de uno en uno con un código como este:

```
// Extender el objeto Range.prototype predefinido para no sobrescribir // la  
propiedad Range.prototype.constructor creada automáticamente. Range.  
prototype.includes = function(x) { return this.from <= x && x <= this.to; }; Range.  
prototype.toString = function() { return "(" + this.from + "..." + this.to + ")"; };
```

## 9.3 Clases con la palabra clave class

Las clases han sido parte de JavaScript desde la primera versión del lenguaje, pero en ES6, finalmente obtuvieron su propia sintaxis con la introducción de la palabra clave class. El ejemplo 9-3 muestra el aspecto de nuestra clase Range cuando se escribe con esta nueva sintaxis.

Ejemplo 9-3. La clase Range reescrita con la clase

---

```

class Rango { constructor(desde, hasta) {
    // Almacena los puntos inicial y final (estado) de este nuevo objeto de rango.      //
    // Estas son propiedades no heredadas que son únicas para este objeto.
    this.from = from; this.to = to;
}

// Devuelve true si x está en el rango, false en caso contrario
// Este método funciona tanto para rangos textuales y de fechas como numéricos.
includes(x) { return this.from <= x && x <= this.to; }

// Una función generadora que hace que las instancias de la clase sean iterables.

// Ten en cuenta que sólo funciona para rangos numéricos. *[Symbol.iterator]()
for(let x = Math. ceil(this.from); x <= this.to; x++) yield x;

// Devuelve una representación en forma de cadena del rango toString() {
devuelve `{$estو. desde}... {$estو. hasta}`; } }

// Aquí hay ejemplos de uso de esta nueva clase Range let r = new
Range(1,3); // Crear un objeto Range
r. includes(2) //=> true: 2 está en el rango
r. toString() //=> "(1...3)"
[... r] //=> [1, 2, 3]; convertir en un array mediante un iterador

```

Es importante entender que las clases definidas en los Ejemplos [9-2](#) y [9-3](#) funcionan exactamente de la misma manera. La introducción de la palabra clave `class` en el lenguaje no altera la naturaleza fundamental de las clases basadas en prototipos de JavaScript. Y aunque el Ejemplo 9-3 utiliza la palabra clave `class`, el objeto `Range` resultante es una función constructora, igual que la versión definida en el Ejemplo 9-2. La nueva sintaxis de clase es

limpia y conveniente, pero es mejor considerarla como "azúcar sintáctico" para el mecanismo de definición de clase más fundamental mostrado en el Ejemplo 9-2.

Observe lo siguiente sobre la sintaxis de la clase en el Ejemplo 9-3:

- La clase se declara con la palabra clave class, que va seguida del nombre de la clase y de un cuerpo de clase entre llaves.
- El cuerpo de la clase incluye definiciones de métodos que utilizan la abreviatura de método literal de objeto (que también utilizamos en el Ejemplo 9-1), donde se omite la palabra clave function. Sin embargo, a diferencia de los literales de objeto, no se utilizan comas para separar los métodos entre sí. (Aunque los cuerpos de las clases son superficialmente similares a los literales de los objetos, no son lo mismo. En particular, no admiten la definición de propiedades con pares nombre/valor).
- La palabra clave constructor se utiliza para definir la función constructora de la clase. Sin embargo, la función definida no se llama realmente "constructor". La declaración de la clase define una nueva variable Range y asigna el valor de esta función constructora especial a esa variable.
- Si su clase no necesita hacer ninguna inicialización, puede omitir la palabra clave constructor y su cuerpo, y una función constructora vacía será creada implícitamente para usted.

Si desea definir una clase que subclase -o *herede*- de otra clase, puede utilizar la palabra clave extends con la palabra clave class:

```
// Un Span es como un Range, pero en lugar de inicializarlo con
// un inicio y un final, lo inicializamos con un inicio y una longitud class Span extends
Range { constructor(inicio, longitud) { if (longitud >= 0) { super(inicio, inicio + longitud); }
else { super(inicio + longitud, inicio); }
}
```

```
}
```

La creación de subclases es un tema completo. Volveremos a él, y explicaremos las palabras clave extends y super mostradas aquí, en [§9.5](#).

Al igual que las declaraciones de funciones, las declaraciones de clases tienen tanto forma de declaración como de expresión. Al igual que podemos escribir

```
let cuadrado = function(x) { return x * x; }; cuadrado(3) //=> 9
```

también podemos escribir:

```
let Cuadrado = class { constructor(x) { this.area = x * x; } }; new Square(3).area //=> 9
```

Al igual que las expresiones de definición de funciones, las expresiones de definición de clases pueden incluir un nombre de clase opcional. Si se proporciona dicho nombre, éste sólo se define dentro del propio cuerpo de la clase.

Aunque las expresiones de función son bastante comunes (en particular con la abreviatura de función de flecha), en la programación de JavaScript, las expresiones de definición de clase no son algo que probablemente utilice mucho, a menos que se encuentre escribiendo una función que toma una clase como su argumento y devuelve una subclase.

Concluiremos esta introducción a la palabra clave `class` mencionando un par de cosas importantes que deberías saber y que no son evidentes en la sintaxis de `class`:

- Todo el código dentro del cuerpo de una declaración de clase está implícitamente en modo estricto ([§5.6.3](#)), incluso si no aparece ninguna directiva "use strict". Esto significa, por ejemplo, que no puede utilizar literales enteros octales o la sentencia with dentro de los cuerpos de las clases y que es más probable que obtenga errores de sintaxis si se olvida de declarar una variable antes de utilizarla.
- A diferencia de las declaraciones de función, las declaraciones de clase no son "elevadas". Recuerde de [§8.1.1](#) que las definiciones de funciones se comportan como si se hubieran movido a la parte superior del archivo o función que las encierra, lo que significa que puede invocar una función en el código que viene antes de la definición real de la función. Aunque las declaraciones de clases son como las declaraciones de funciones en algunos aspectos, no comparten este comportamiento de elevación: *no se puede* instanciar una clase antes de declararla.

### 9.3.1 Métodos estáticos

Puede definir un método estático dentro del cuerpo de una clase anteponiendo a la declaración del método la palabra clave static. Los métodos estáticos se definen como propiedades de la función constructora en lugar de propiedades del objeto prototipo.

Por ejemplo, supongamos que añadimos el siguiente código al

#### Ejemplo 9-3:

```
static parse(s) { let matches = s. match(/^\((\d+)\.\.(\d+)\)$/); if (!
matches) { throw new TypeError(`No se puede analizar el rango de
"${s}"`); } return new Range(parseInt(matches[1]), parseInt(matches[2]));
}
```

El método definido por este código es Range.parse(), no

`Range.prototype.parse()`, y debes invocarlo a través del constructor, no a través de una instancia:

```
let r = Range.parse('1...10'); // Devuelve un nuevo objeto Range  
r.parse('1...10'); // TypeError: r.parse no es una función
```

A veces verás que los métodos estáticos se llaman métodos *de clase* porque se invocan usando el nombre de la clase/constructor. Cuando se utiliza este término, es para contrastar los métodos de clase con los *métodos de instancia* regulares que se invocan en las instancias de la clase. Debido a que los métodos estáticos son invocados en el constructor y no en una instancia en particular, casi nunca tiene sentido usar la palabra clave `this` en un método estático.

Veremos ejemplos de métodos estáticos en [el Ejemplo 9-4](#).

### 9.3.2 Getters, Setters y otras formas de métodos

Dentro del cuerpo de una clase, puedes definir métodos `getter` y `setter` ([§6.10.6](#)) igual que en los literales de los objetos. La única diferencia es que en los cuerpos de clase, no se pone una coma después del `getter` o `setter`. [El ejemplo 9-4](#) incluye un ejemplo práctico de un método `getter` en una clase.

En general, todas las sintaxis de definición de métodos abreviados permitidas en los literales de los objetos también se permiten en los cuerpos de las clases. Esto incluye los métodos generadores (marcados con `*`) y los métodos cuyos nombres son el valor de una expresión entre corchetes. De hecho, ya has visto (en el [Ejemplo 9-3](#)) un método generador con un nombre calculado que hace que la clase `Range` sea iterable:

```
*[Symbol.iterator]() { for(let x = Math.ceil(this.from); x <= this.to; x++) yield x;
}
```

### 9.3.3 Campos públicos, privados y estáticos

En la discusión aquí de las clases definidas con la palabra clave `class`, sólo hemos descrito la definición de métodos dentro del cuerpo de la clase. El estándar ES6 sólo permite la creación de métodos (incluyendo getters, setters y generadores) y métodos estáticos; no incluye la sintaxis para definir campos. Si quieres definir un campo (que no es más que un sinónimo orientado a objetos de "propiedad") en una instancia de clase, debes hacerlo en la función constructora o en uno de los métodos. Y si quiere definir un campo estático para una clase, debe hacerlo fuera del cuerpo de la clase, después de que la clase haya sido definida. [El ejemplo 9-4](#) incluye ejemplos de ambos tipos de campos.

Sin embargo, se está estandarizando la sintaxis de clase extendida que permite la definición de campos de instancia y estáticos, tanto en forma pública como privada. El código que se muestra en el resto de esta sección todavía no es estándar de JavaScript a principios de 2020, pero ya es compatible con Chrome y parcialmente con Firefox (solo campos de instancia públicos). La sintaxis de los campos de instancia públicos es de uso común por parte de los programadores de JavaScript que utilizan el framework React y el transpilador Babel.

Suponga que está escribiendo una clase como ésta, con un constructor que inicializa tres campos:

```
class Buffer { constructor() { this.size = 0; this.capacity = 4096; this.buffer = new
Uint8Array(this.capacity); }
}
```

Con la nueva sintaxis de los campos de instancia que probablemente se estandarice, se podría escribir en su lugar:

```
class Buffer { size = 0; capacity = 4096; buffer = new  
Uint8Array(this.capacity); }
```

El código de inicialización del campo se ha movido fuera del constructor y ahora aparece directamente en el cuerpo de la clase. (Ese código se sigue ejecutando como parte del constructor, por supuesto. Si no se define un constructor, los campos se inicializan como parte del constructor creado implícitamente). Los prefijos this. que aparecían en el lado izquierdo de las asignaciones han desaparecido, pero tenga en cuenta que todavía debe utilizar this. para referirse a estos campos, incluso en el lado derecho de las asignaciones del inicializador. La ventaja de inicializar los campos de instancia de esta manera es que esta sintaxis permite (pero no requiere) poner los inicializadores en la parte superior de la definición de la clase, dejando claro a los lectores exactamente qué campos mantendrán el estado de cada instancia. Puedes declarar campos sin un inicializador escribiendo simplemente el nombre del campo seguido de un punto y coma. Si haces eso, el valor inicial del campo será indefinido. Es mejor estilo hacer siempre el valor inicial explícito para todos los campos de su clase.

Antes de añadir esta sintaxis de campo, los cuerpos de las clases se parecían mucho a los literales de los objetos que utilizaban la sintaxis de los métodos abreviados, salvo que se habían eliminado las comas. Esta sintaxis de campo -con signos de igualdad y punto y coma en lugar de dos puntos y comas- deja claro que los cuerpos de las clases no son en absoluto iguales a los literales de los objetos.

La misma propuesta que pretende estandarizar estos campos de instancia también define los campos de instancia privados. Si utilizas la sintaxis de inicialización de campos de instancia mostrada en el ejemplo anterior para definir un campo cuyo nombre comienza con # (que normalmente no es un carácter legal en los identificadores de JavaScript), ese campo será utilizable (con el prefijo #) dentro del cuerpo de la clase pero será invisible e inaccesible (y por tanto immutable) para cualquier código fuera del cuerpo de la clase. Si, para la clase Buffer hipotética anterior, quisieras asegurarte de que los usuarios de la clase no pudieran modificar inadvertidamente el campo de tamaño de una instancia, podrías utilizar un campo privado #size en su lugar, y luego definir una función getter para proporcionar acceso de sólo lectura al valor:

```
class Buffer { #size = 0; get size() { return this.  
#tamaño; } }
```

Tenga en cuenta que los campos privados deben ser declarados utilizando esta nueva sintaxis de campo antes de que puedan ser utilizados. No se puede escribir this.#size = 0; en el constructor de una clase a menos que se incluya una "declaración" del campo directamente en el cuerpo de la clase.

Por último, una propuesta relacionada pretende normalizar el uso de la palabra clave static para los campos. Si añades static antes de una declaración de campo público o privado, esos campos se crearán como propiedades de la función constructora en lugar de propiedades de las instancias. Considere el método estático Range.parse() que hemos definido. Incluye una expresión regular bastante compleja que podría ser bueno factorizar en su propio

campo estático. Con la nueva sintaxis de campo estático propuesta, podríamos hacerlo así:

```
static integerRangePattern = /\^((\d+)\.(\d+))$/; static parse(s) { let
matches = s. match(Range. integerRangePattern); if (! matches) { throw new
TypeError(`No se puede analizar Range de "${s}"`) } return new
Range(parseInt(matches[1]), matches[2]); }
```

Si quisieramos que este campo estático fuera accesible sólo dentro de la clase, podríamos hacerlo privado usando un nombre como #patrón.

### 9.3.4 Ejemplo: Una clase de números complejos

El [ejemplo 9-4](#) define una clase para representar números complejos. La clase es relativamente simple, pero incluye métodos de instancia (incluyendo getters), métodos estáticos, campos de instancia y campos estáticos. Incluye algo de código comentado que demuestra cómo podríamos utilizar la sintaxis aún no estándar para definir campos de instancia y campos estáticos dentro del cuerpo de la clase.

#### Ejemplo 9-4. Complex.js: una clase de números complejos

---

```
/** * Las instancias de esta clase Complex representan números complejos.
* Recordemos que un número complejo es la suma de un número real y un * número
imaginario y que el número imaginario i es la raíz cuadrada de -1. */ clase Complejo {
    // Una vez estandarizadas las declaraciones de los campos de la clase, podríamos declarar
    // campos privados para mantener las partes real e imaginaria de un
```

número complejo

// aquí, con un código como este:

//  
// #r = 0;  
// #i = 0;

// Esta función constructora define los campos de instancia r e i en cada instancia que crea. Estos campos contienen las partes real e imaginaria del número complejo: son el estado del objeto.

```
constructor(real, imaginario) { this.r = real; //  
Este campo contiene la parte real del número.  
this.i = imaginary; // Este campo contiene la parte imaginaria. }
```

// Aquí hay dos métodos de instancia para la suma y la multiplicación de los números complejos. Si c y d son instancias de esta clase, podríamos escribir c.plus(d) o d.times(c)

```
plus(that) { devuelve new Complex(this.r + that.r, this.i + that.i); }  
times(that) { devuelve new Complex(this.r * that.r - this.i * that.i, this.r * that.i + this.i * that.r); }
```

// Y aquí están las variantes estáticas de los métodos aritméticos complejos. //

Podríamos escribir Complex.sum(c,d) y Complex.product(c,d)

```
static sum(c, d) { return c.plus(d); } static product(c, d) { return c.times(d); }
```

// Estos son algunos métodos de instancia que se definen como getters // para que se usen como campos. Los getters reales e imaginarios serían // sería útil si usáramos campos privados this.#r y this.#i

```
get real() { return this.r; }
```

```

get imaginary() { return this.i; } get magnitude() { return Math.hypot(this.r,
this.i); }

// Las clases deberían tener casi siempre un método toString() toString() { return `{{this.r},${this.i}}`;

// A menudo es útil definir un método para comprobar si // dos instancias de su
clase representan el mismo valor equals(that) { return that instanceof Complex &&
this.r === that.r && this.i === that.i; }

// Una vez que los campos estáticos son soportados dentro de los cuerpos de las clases,
podríamos
// defina una constante Complex.ZERO útil como esta:
// static ZERO = new Complex(0,0);
}

// Aquí hay algunos campos de clase que contienen útiles números complejos
predefinidos. Complejo.ZERO = new Complex(0,0);
Complejo.UNO = nuevo Complejo(1,0);
Complejo.I = nuevo Complejo(0,1);

```

Con la clase Complex del [Ejemplo 9-4](#) definida, podemos utilizar el constructor, los campos de instancia, los métodos de instancia, los campos de clase y los métodos de clase con un código como este:

```

let c = new Complex(2, 3); // Crear un nuevo objeto con el constructor let d = new
Complex(c, i, c.r); // Utilizar los campos de instancia de c
c.plus(d).toString() // => "5,5"; utilizar métodos de instancia
c.magnitud // => Math.hypot(2,3); utiliza una función getter Complex.product(c, d)
// => new Complex(0, 13); un método estático Complex.ZERO.toString() // =>
"{0,0}"; una propiedad estática

```

## 9.4 Añadir métodos a las clases existentes

El mecanismo de herencia basado en prototipos de JavaScript es dinámico: un objeto hereda propiedades de su prototipo, incluso si las propiedades del prototipo cambian después de la creación del objeto. Esto significa que podemos aumentar las clases de

JavaScript simplemente añadiendo nuevos métodos a sus objetos prototipo.

Aquí, por ejemplo, está el código que añade un método para calcular el complejo conjugado a la clase Complex del [Ejemplo 9-4](#):

```
// Devuelve un número complejo que es el conjugado complejo de éste. Complex.  
prototype.conj = function() { return new  
Complex(this.r, -this.i); };
```

El objeto prototipo de las clases incorporadas de JavaScript también está abierto de esta manera, lo que significa que podemos añadir métodos a números, cadenas, matrices, funciones, etc. Esto es útil para implementar nuevas características del lenguaje en versiones antiguas del mismo:

```
// Si el nuevo método String startsWith() no está ya definido...  
if (!String.prototype.startsWith) {  
    // ...entonces defínelo así usando el método antiguo indexOf(). String.prototype.  
startsWith = function(s) { return this.indexOf(s) === 0; };  
}
```

He aquí otro ejemplo:

```
// Invocar la función f tantas veces, pasando el número de iteración  
// Por ejemplo, para imprimir "hola" 3 veces:  
// deja que n = 3;  
// n.times(i => { console.log(`hola ${i}`); }); Number.prototype.times = function(f,  
context) { let n = this.valueOf();  
    for(let i = 0; i < n; i++) f.call(context, i); };
```

Añadir métodos a los prototipos de los tipos incorporados de esta manera se considera generalmente una mala idea porque causará confusión y problemas de compatibilidad en el futuro si una nueva versión de JavaScript define un método con el mismo nombre.

Incluso es posible añadir métodos a `Object.prototype`, haciéndolos disponibles para todos los objetos. Pero esto nunca es bueno porque las propiedades añadidas a `Object.prototype` son visibles para los bucles `for/in` (aunque se puede evitar esto utilizando `Object.defineProperty()` [§14.1] para que la nueva propiedad no sea enumerable).

## 9.5 Subclases

En la programación orientada a objetos, una clase B puede *extender* o *subclasificar* otra clase A. Decimos que A es la *superclase* y B es la *subclase*. Las instancias de B heredan los métodos de A. La clase B puede definir sus propios métodos, algunos de los cuales pueden *anular* los métodos del mismo nombre definidos por la clase A. Si un método de B anula un método de A, el método que anula en B a menudo necesita invocar el método anulado en A. Del mismo modo, el constructor de la subclase B() normalmente debe invocar el constructor de la superclase A() con el fin de garantizar que las instancias están completamente inicializadas.

Esta sección comienza mostrando cómo definir subclases de la manera antigua, anterior a la ES6, y luego pasa rápidamente a demostrar la subclasicación utilizando las palabras clave `class` y `extends` y la invocación del método `constructor` de la superclase con la palabra clave `super`. A continuación hay una subsección sobre cómo evitar las subclases y confiar en la composición de objetos en lugar de la herencia. La sección termina con un ejemplo extendido que define una jerarquía de clases `Set` y demuestra

cómo las clases abstractas pueden ser utilizadas para separar la interfaz de la implementación.

### 9.5.1 Subclases y prototipos

Supongamos que queremos definir una subclase `Span` de la clase `Range` del [Ejemplo 9-2](#). Esta subclase funcionará como un `Range`, pero en lugar de inicializarla con un inicio y un final, especificaremos un inicio y una distancia, o `span`. Una instancia de esta clase `Span` es también una instancia de la superclase `Range`. Una instancia de `Span` hereda un método personalizado `toString()` de `Span.prototype`, pero para ser una subclase de `Range`, también debe heredar métodos (como `includes()`) de `Range.prototype`.

#### *Ejemplo 9-5. Span.js: una simple subclase de Range*

---

```
// Esta es la función constructora de nuestra subclase function Span(start, span)
{ if (span >= 0) { this. from = start; this. to = start + span; } else { this. to = start;
this. from = start + span; }
}

// Asegúrese de que el prototipo Span hereda del prototipo Range Span.prototype
// = Object. create(Range.prototype);

// No queremos heredar Range.prototype.constructor, así que // definimos nuestra propia
// propiedad constructora.
Span.prototype.constructor = Span;

// Al definir su propio método toString(), Span anula el método // toString() que de
// otro modo heredaría de Range. Span.prototype.toString = function() {
    return `{$este. desde}... +${este. hasta - este. desde}`;
};
```

Para que `Span` sea una subclase de `Range`, tenemos que hacer que `Span.prototype` herede de `Range.prototype`. La clave

Línea de código en el ejemplo anterior es ésta, y si tiene sentido para ti, entiendes cómo funcionan las subclases en JavaScript:

```
Span.prototype = Object.create(Range.prototype);
```

Los objetos creados con el constructor `Span()` heredarán de la familia objeto `Span.prototype`. Pero hemos creado ese objeto para que herede de `Range.prototype`, por lo que los objetos `Span` heredarán tanto de `Span.prototype` como de `Range.prototype`.

Puede notar que nuestro constructor `Span()` establece las mismas propiedades `from` y `to` que el constructor `Range()` y por lo tanto no necesita invocar el constructor `Range()` para inicializar el nuevo objeto. Del mismo modo, el método `toString()` de `Span` reimplementa completamente la conversión de cadenas sin necesidad de llamar a la versión de `Range` de `toString()`. Esto hace que `Span` sea un caso especial, y sólo podemos salirnos con la nuestra con este tipo de subclasiación porque conocemos los detalles de implementación de la superclase. Un mecanismo robusto de subclasiación necesita permitir que las clases invoquen los métodos y el constructor de su superclase, pero antes de ES6, JavaScript no tenía una forma sencilla de hacer estas cosas.

Afortunadamente, ES6 resuelve estos problemas con la palabra clave `super` como parte de la sintaxis de la clase. La siguiente sección demuestra cómo funciona.

## 9.5.2 Subclases con extends y super

En ES6 y posteriores, se puede crear una superclase simplemente añadiendo una cláusula extends a la declaración de una clase, y se puede hacer esto incluso para las clases incorporadas:

```
// Una subclase trivial de Array que añade getters para el primer y el último elemento.
class EZArray extends Array { get first() { return this[0]; } get last()
{ return this[this.length-1]; } }

let a = new EZArray(); a instanceof EZArray // => true: a es una instancia de la
subclase a instanceof Array // => true: a es también una instancia de la
superclase.
a.push(1,2,3,4); // a.length == 4; podemos usar métodos heredados
a.pop() // => 4: otro método heredado
a.first // => 1: primer getter definido por la subclase
a.last // => 3: último getter definido por la subclase a[1] // => 2: la sintaxis regular de
acceso a arrays sigue funcionando. Array.isArray(a) // => true: la instancia de la
subclase es realmente un array
EZArray.isArray(a) // => true: la subclase hereda la estática
```

métodos, también!

Esta subclase de EZArray define dos métodos getter simples. Las instancias de EZArray se comportan como matrices ordinarias, y podemos utilizar los métodos y propiedades heredadas como push(), pop() y length. Pero también podemos utilizar los primeros y últimos getters definidos en la subclase. No sólo se heredan métodos de instancia como pop(), sino que también se heredan métodos estáticos como Array.isArray. Esta es una nueva

característica habilitada por la sintaxis de clases de ES6: EZArray() es una función, pero hereda de Array():

```
// EZArray hereda los métodos de instancia porque  
EZArray.prototype  
// hereda de Array.prototype  
Array.prototype.isPrototypeOf(EZArray.prototype) //=> true  
  
// Y EZArray hereda métodos y propiedades estáticas porque // EZArray hereda de  
Array. Esta es una característica especial de la palabra clave // extends y no es posible  
antes de ES6.  
Array.isPrototypeOf(EZArray) //=> true
```

Nuestra subclase EZArray es demasiado simple para ser muy instructiva. El ejemplo [9-6](#) es un ejemplo más completo. Define una subclase TypedMap de la clase Map incorporada que añade comprobación de tipos para asegurar que las claves y los valores del mapa son de los tipos especificados (según typeof). Es importante destacar que este ejemplo demuestra el uso de la palabra clave super para invocar el constructor y los métodos de la superclase.

*Ejemplo 9-6. TypedMap.js: una subclase de Map que comprueba los tipos de clave y valor*

---

```
class TypedMap extends Map {
```

```
constructor(keyType, valueType, entries) { // Si se especifican entradas, comprueba sus tipos
  if (entries) {
    for(let [k, v] of entries) {
      if (typeof k !== keyType || typeof v !== valueType)
        throw new TypeError(`Tipo incorrecto para la entrada [${k}, ${v}]`);
    }
  }
}

// Inicializar la superclase con las entradas iniciales (de tipo comprobado)
super(entradas);

// Y luego inicializar esta subclase almacenando los tipos
this.keyType = keyType;
this.valueType = valueType;

// Ahora redefine el método set() para añadir la comprobación de tipo para cualquier nueva entrada añadida al mapa.
set(key, value) {
  // Lanza un error si la clave o el valor son del tipo incorrecto
  if (this.keyType && typeof key !== this.keyType)
    lanza un nuevo TypeError(`#${key} no es del tipo ${this.keyType}`);
  if (this.valueType && typeof value !== this.valueType)
    lanza un nuevo TypeError(`#${value} no es del tipo ${this.valueType}`);
}

// Si los tipos son correctos, invocamos la versión de la superclase de el método set(),
// para añadir realmente la entrada al mapa. Y nosotros
```

```
// devuelve lo que el método de la superclase devuelva.  
return super.set(key, value); }  
}
```

Los dos primeros argumentos del constructor TypedMap() son los tipos de clave y valor deseados. Deben ser cadenas, como "number" y "boolean", que el operador typeof devuelve. También puede especificar un tercer argumento: un array (o cualquier objeto iterable) de arrays [key,value] que especifique las entradas iniciales del mapa. Si especifica alguna entrada inicial, lo primero que hace el constructor es verificar que sus tipos son correctos. A continuación, el constructor invoca al constructor de la superclase, utilizando la palabra clave super como si fuera un nombre de función. El constructor Map() toma un argumento opcional: un objeto iterable de arrays [clave,valor]. Así que el tercer argumento opcional del constructor TypedMap() es el primer argumento opcional del constructor Map(), y lo pasamos a ese constructor de la superclase con super(entradas).

Después de invocar el constructor de la superclase para inicializar el estado de la superclase, el constructor de TypedMap() inicializa su propio estado de la subclase estableciendo this.keyType y this.valueType a los valores especificados tipos. Necesita establecer estas propiedades para poder utilizarlas de nuevo en el método set().

Hay algunas reglas importantes que debes conocer sobre el uso de super() en los constructores:

- Si defines una clase con la palabra clave extends, entonces el constructor de tu clase debe usar super() para invocar el constructor de la superclase.
- Si no defines un constructor en tu subclase, se definirá uno automáticamente por ti. Este constructor definido implícitamente simplemente toma cualquier valor que se le pase y pasa esos valores a super().
- No puede utilizar la palabra clave this en su constructor hasta que haya invocado el constructor de la superclase con super(). Esto refuerza la regla de que las superclases se inicializan a sí mismas antes que las subclases.
- La expresión especial new.target es indefinida en las funciones que se invocan sin la palabra clave new. En las funciones constructoras, sin embargo, new.target es una referencia al constructor que fue invocado. Cuando se invoca un constructor de una subclase y se utiliza super() para invocar el constructor de la superclase, ese constructor de la superclase verá el constructor de la subclase como el valor de new.target. Una superclase bien diseñada no debería necesitar saber si ha sido subclasicada, pero podría ser útil poder utilizar new.target.name en los mensajes de registro, por ejemplo.

Después del constructor, la siguiente parte del [Ejemplo 9-6](#) es un método llamado set(). La superclase Map define un método llamado set() para añadir una nueva entrada al mapa. Decimos que este método set() de TypedMap *anula* el método set() de su superclase. Esta simple subclase de TypedMap no sabe nada sobre cómo añadir nuevas entradas al mapa, pero sí sabe cómo comprobar los tipos, así que eso es lo que hace primero, verificando que la clave y el valor a añadir al mapa tienen los tipos

correctos y lanzando un error si no es así. Este método set() no tiene ninguna forma de añadir la clave y el valor al propio mapa, pero para eso está el método set() de la superclase. Así que usamos la palabra clave super de nuevo para invocar la versión del método de la superclase. En este contexto, super funciona de forma muy parecida a la palabra clave this: se refiere al objeto actual pero permite acceder a los métodos anulados definidos en la superclase.

En los constructores, se requiere invocar el constructor de la superclase antes de poder acceder a éste e inicializar el nuevo objeto por sí mismo. No existen estas reglas cuando se sobrescribe un método. Un método que sobrescribe un método de la superclase no está obligado a invocar el método de la superclase. Si utiliza super para invocar el método sobrescrito (o cualquier método) de la superclase, puede hacerlo al principio, en medio o al final del método sobrescrito.

Por último, antes de dejar el ejemplo de TypedMap, cabe destacar que esta clase es un candidato ideal para el uso de campos privados. Tal y como está escrita la clase ahora, un usuario podría cambiar las propiedades keyType o valueType para subvertir la comprobación de tipos. Una vez que se soporten los campos privados, podríamos cambiar estas propiedades a #keyType y #valueType para que no puedan ser alteradas desde el exterior.

### 9.5.3 Delegación en lugar de herencia

La palabra clave extends facilita la creación de subclases. Pero eso no significa que *debas* crear muchas subclases. Si quieres escribir una clase que comparta el comportamiento de alguna otra clase,

puedes intentar heredar ese comportamiento creando una subclase. Pero a menudo es más fácil y más flexible conseguir ese comportamiento deseado en tu clase haciendo que tu clase cree una instancia de la otra clase y simplemente delegando en esa instancia cuando sea necesario. No se crea una nueva clase subclasiificando, sino envolviendo o "componiendo" otras clases. Este enfoque de delegación se llama a menudo "composición", y es una máxima muy citada de la programación orientada a objetos que uno debe "favorecer la composición sobre la herencia".<sup>2</sup>

Supongamos, por ejemplo, que queremos una clase Histograma que se comporte como la clase Set de JavaScript, excepto que en lugar de llevar la cuenta de si un valor se ha añadido a set o no, mantiene un recuento del número de veces que se ha añadido el valor. Dado que la API de esta clase Histograma es similar a la de Set, podríamos considerar subclasificar Set y añadir un método count(). Por otro lado, una vez que empecemos a pensar en cómo podríamos implementar este método count(), podríamos darnos cuenta de que la clase Histograma es más parecida a Map que a Set porque necesita mantener un mapeo entre los valores y el número de veces que han sido añadidos. Así que en lugar de subclasificar Set, podemos crear una clase que defina una API similar a Set pero que implemente esos métodos delegando en un objeto Map interno.

El ejemplo 9-7 muestra cómo podemos hacer esto.

*Ejemplo 9-7. Histograma.js: una clase tipo Set implementada con delegación*

---

```
/**
```

```
* Una clase similar a Set que lleva la cuenta de las veces que un valor tiene
```

\* se ha añadido. Llame a add() y remove() como lo haría para un  
Set, y

- \* llamar a count() para saber cuántas veces se ha sumado un valor determinado.
- \* El iterador por defecto devuelve los valores que han sido añadidos al menos \*  
una vez. Utilice entries() si desea iterar pares [valor, cuenta]. \*/

```
class Histogram {
```

```

// Para inicializar, simplemente creamos un objeto Map para delegar en constructor()
{this. map = new Map();}

// Para cualquier clave dada, el recuento es el valor en el Mapa, o cero // si la clave no
aparece en el Mapa. count(key) { return this. map. get(key) || 0; }

// El método tipo Set has() devuelve true si la cuenta es distinta de cero has(key) {
return this. count(key) > 0; }

// El tamaño del histograma es sólo el número de entradas en el Mapa. get size() {
return this. map. size; }

// Para añadir una clave, basta con incrementar su cuenta en el Mapa. add(key) { this.
map. set(key, this. count(key) + 1); }

// Borrar una clave es un poco más complicado porque tenemos que borrar // la clave
del Mapa si la cuenta vuelve a ser cero.
delete(key) { let count = this. count(key); if (count === 1)
{ this. map. delete(key); } else if (count > 1) { this. map.
set(key, count - 1); }
}

// Iterar un histograma sólo devuelve las claves almacenadas en él
[Symbol. iterator]() { devuelve this. map. keys(); }

// Estos otros métodos del iterador sólo delegan en el objeto Map keys() { devuelve
this. map. keys(); } values() { devuelve this. map. values(); } entries() { devuelve this.
map. entries(); }
}

```

Todo lo que hace el constructor Histogram() en el [Ejemplo 9-7](#) es crear un objeto Map. Y la mayoría de los métodos son de una sola línea que simplemente delegan a un método del mapa, haciendo la

implementación bastante simple. Debido a que usamos la delegación en lugar de la herencia, un objeto Histograma no es una instancia de Set o Map. Pero Histogram implementa un número de métodos de Set comúnmente utilizados, y en un lenguaje no tipado como JavaScript, eso es a menudo suficiente: una relación de herencia formal es a veces agradable, pero a menudo opcional.

#### 9.5.4 Jerarquías de clases y clases abstractas

El Ejemplo [9-6](#) demostró cómo podemos subclásificarnos Map. El Ejemplo [9-7](#) demostró cómo podemos delegar en un objeto Map sin subclásificarnos nada. Usar clases de JavaScript para encapsular datos y modularizar su código es a menudo una gran técnica, y usted puede encontrarse usando la palabra clave class frecuentemente. Pero puedes encontrarte con que prefieres la composición a la herencia y que raramente necesitas usar extends (excepto cuando estás usando una librería o framework que requiere que extiendas sus clases base).

Sin embargo, hay algunas circunstancias en las que son apropiados múltiples niveles de subclases, y terminaremos este capítulo con un ejemplo extendido que demuestra una jerarquía de clases que representan diferentes tipos de conjuntos. (Las clases de conjuntos definidas en el Ejemplo [9-8](#) son similares, pero no completamente compatibles, con la clase Set incorporada de JavaScript).

El ejemplo [9-8](#) define muchas subclases, pero también demuestra cómo se pueden definir *clases abstractas* -clases que no incluyen una implementación completa- para que sirvan de superclase común para un grupo de subclases relacionadas. Una superclase

abstracta puede definir una implementación parcial que todas las subclases heredan y comparten. Las subclases, entonces, sólo necesitan definir su propio comportamiento único implementando los métodos abstractos definidos -pero no implementados- por la superclase. Tenga en cuenta que JavaScript no tiene ninguna definición formal de métodos abstractos o clases abstractas; simplemente estoy usando ese nombre aquí para los métodos no implementados y las clases incompletamente implementadas.

El ejemplo 9-8 está bien comentado y se sostiene por sí mismo. Le animo a que lo lea como un ejemplo final de este capítulo sobre clases. La clase final en el Ejemplo 9-8 hace mucha manipulación de bits con los operadores &, | y ~, que puede revisar en §4.8.3.

*Ejemplo 9-8. Sets.js: una jerarquía de clases de conjuntos abstractos y concretos*

---

```
/**  
 *      La clase AbstractSet define un único método abstracto, has(). */ class AbstractSet  
{  
    // Lanza un error aquí para que las subclases sean forzadas  
    // para definir su propia versión de trabajo de este método.  
    has(x) { throw new Error("Método abstracto"); } }  
  
/** * NotSet es una subclase concreta de AbstractSet.  
 *      Los miembros de este conjunto son todos los valores que no son miembros de  
algún  
*      otro conjunto. Como está definido en términos de otro conjunto, no es * escribible,  
y como tiene infinitos miembros, no es enumerable.  
*      Lo único que podemos hacer con él es comprobar la pertenencia y convertirlo en  
un  
*      cadena utilizando la notación matemática.  
*/ class NotSet extends AbstractSet {
```

```

constructor(set) { super(); this.
set = set; }

// Nuestra implementación del método abstracto que heredamos has(x) { return ! this.
set. has(x); } // Y también anulamos este método Object toString() { return '{ x| x ∈
${this. set. toString()} }'; }

}

/**
El conjunto de rango es una subclase concreta de AbstractSet. Sus miembros son * todos los valores que se encuentran entre los límites desde y hasta, inclusive. Dado que sus miembros pueden ser números de coma flotante, no es * enumerable y no tiene un tamaño significativo.
 */ class RangeSet extends AbstractSet {
constructor(from, to) { super(); this. from = from;
this. to = to; }

has(x) { return x >= this. from && x <= this. to; } toString() { return '{ x| ${this. from} ≤ x ≤
${this. to} }'; }

}

/*
*      AbstractEnumerableSet es una subclase abstracta de
AbstractSet. Define
*      un getter abstracto que devuelve el tamaño del conjunto y también define un
*      iterador abstracto. Y luego implementa los concretos isEmpty(), toString(),
*      y equals(). Las subclases que implementan el método
*      el iterador, el getter de tamaño y el método has() obtienen estos
*      métodos de forma gratuita.
*/

```

```

class AbstractEnumerableSet extends AbstractSet { get size() { throw new Error("Método abstracto"); } [Symbol.iterator]() { throw new Error("Método abstracto"); }

isEmpty() { return this.size === 0; } toString() { return `[${Array.from(this).join(", ")}]`; }
} equals(set) {
    // Si el otro conjunto no es también Enumerable, no es igual a este
    if (!(set instanceof AbstractEnumerableSet)) return false;

    // Si no tienen el mismo tamaño, no son iguales if (this.size !== set.size) return false;

    // Recorre los elementos de este conjunto for(let element of this) {
        // Si un elemento no está en el otro conjunto, no son iguales if (!set.has(element)) return false;

        // Los elementos coinciden, por lo que los conjuntos son iguales return true;
    }

/*
SingletonSet es una subclase concreta de AbstractEnumerableSet. Un conjunto singleton es un conjunto de sólo lectura con un único miembro.
 */ class SingletonSet extends AbstractEnumerableSet {
constructor(member) { super(); this.member = member; }

// Implementamos estos tres métodos, y heredamos las implementaciones de isEmpty, equals() // y toString() basadas en estos métodos. has(x) { return x === this.member; } get size() { return 1; }

```

```
*[Symbol.iterator]() { yield this.member; }
}

/*
*      AbstractWritableSet es una subclase abstracta de AbstractEnumerableSet.
*      Define los métodos abstractos insert() y remove() que insertan y
*      eliminar elementos individuales del conjunto, y luego implementa concretamente
*      los métodos add(), subtract(), y intersect() encima de esos.
Tenga en cuenta que
*      nuestra API difiere aquí de la clase Set estándar de JavaScript. */ class
AbstractWritableSet extends AbstractEnumerableSet { insert(x) { throw new
Error("Método abstracto"); } remove(x) { throw new Error("Método abstracto"); }

add(set) { for(let element of set) { this.
insert(element); } }

subtract(set) { for(let element of set) { this.
remove(element); } }

intersect(set) { for(let element of this) { if (!set.
has(element)) { this.remove(element); }
}
}
}

/**
Un BitSet es una subclase concreta de AbstractWritableSet con
```

```

a
*      implementación de conjuntos de tamaño fijo muy eficiente para conjuntos cuya
*      los elementos son enteros no negativos menores que algún tamaño máximo. */
class BitSet extends AbstractWritableSet { constructor(max) { super(); this. max = max; //
El entero máximo que podemos almacenar. this. n = 0; // Cuántos enteros hay en el
conjunto this. numBytes = Math. floor(max / 8) + 1; // Cuántos bytes necesitamos this.
data = new Uint8Array(this. numBytes); // Los bytes }

// Método interno para comprobar si un valor es un miembro legal de este conjunto
_valid(x) { return Number. isInteger(x) && x >= 0 && x <= this. max; }

// Comprueba si el bit especificado del byte especificado de nuestro // array de datos está
establecido o no. Devuelve verdadero o falso.
_has(byte, bit) { return (this. data[byte] &
BitSet. bits[bit]) !== 0; }

// ¿Está el valor x en este BitSet? has(x) { if (this.
_valid(x)) { let byte = Math. floor(x / 8); let bit = x % 8;
return this. _has(byte, bit); } else { return false; }
}

// Inserta el valor x en el BitSet insert(x) { if (this. _valid(x)) { // Si el valor es válido let
byte = Math. floor(x / 8); // convertir a byte
}
}

```

```
y bit let bit = x % 8; if (!this._has(byte, bit)) { // Si ese bit no está establecido todavía
this.data[byte] |= BitSet.bits[bit]; // entonces establecélo this.n++; // e incrementa el
tamaño del conjunto } } else { throw new TypeError("Elemento de conjunto no
válido: " + x); }

remove(x) { if (this._valid(x)) { // Si el valor es válido let byte = Math.floor(x / 8); //
calcula el byte y el bit let bit = x % 8; if (this._has(byte, bit)) { // Si ese bit ya está
establecido this.data[byte] &= BitSet.masks[bit]; // entonces desactívalo this.n--; // y
disminuye el tamaño } } else { throw new TypeError("Elemento de conjunto no
válido: " + x); }

}

// Un getter para devolver el tamaño del conjunto getSize() {
return this.n;
}

// Iterar el conjunto comprobando sólo cada bit por turno.
//(Podríamos ser mucho más inteligentes y optimizar esto sustancialmente)
*[Symbol.iterator]() { for(let i = 0; i <= this.max; i++) { if (this.has(i)) { yield i;
}

}

}

}
```

```
}

// Algunos valores precalculados utilizados por los métodos has(), insert() y remove()
BitSet.bits = new Uint8Array([1, 2, 4, 8, 16, 32, 64, 128]);
BitSet.masks = new Uint8Array([~1, ~2, ~4, ~8, ~16, ~32, ~64,
~128]);
```

## 9.6 Resumen

En este capítulo se han explicado las principales características de las clases de JavaScript:

- Los objetos que son miembros de la misma clase heredan propiedades del mismo objeto prototipo. El objeto prototipo es la característica clave de las clases de JavaScript, y es posible definir clases con nada más que el método `Object.create()`.
- Antes de ES6, las clases se definían más típicamente definiendo primero una función constructora. Las funciones creadas con la palabra clave `function` tienen una propiedad prototipo, y la propiedad El valor de esta propiedad es un objeto que se utiliza como prototipo de todos los objetos creados cuando se invoca la función con `new` como constructor. Al inicializar este objeto prototipo, se pueden definir los métodos compartidos de la clase. Aunque el objeto prototipo es la característica clave de la clase, la función constructora es la identidad pública de la clase.
- ES6 introduce una palabra clave `class` que facilita la definición de clases, pero bajo el capó, el mecanismo de constructores y prototipos sigue siendo el mismo.
- Las subclases se definen mediante la palabra clave `extends` en la declaración de una clase.

- Las subclases pueden invocar el constructor de su superclase o los métodos sobrescritos de su superclase con la palabra clave `super`.
- 

Excepto las funciones devueltas por el método ES5 `Function.bind()`. Funciones enlazadas<sup>1</sup> no tienen propiedad de prototipo propia, pero utilizan el prototipo de la función subyacente si se invocan como constructores.

Véase *Design Patterns* (Addison-Wesley Professional) de Erich Gamma et al. o *Effective Java*<sup>2</sup> (Addison-Wesley Professional) de Joshua Bloch, por ejemplo.



# Capítulo 10. Módulos

---

El objetivo de la programación modular es permitir que los programas grandes se ensamblen utilizando módulos de código de autores y fuentes dispares y que todo ese código se ejecute correctamente incluso en presencia de código que los diversos autores de los módulos no previeron. En la práctica, la modularidad consiste sobre todo en encapsular u ocultar los detalles de implementación privados y mantener el espacio de nombres global ordenado para que los módulos no puedan modificar accidentalmente las variables, funciones y clases definidas por otros módulos.

Hasta hace poco, JavaScript no tenía soporte incorporado para módulos, y los programadores que trabajaban en grandes bases de código hacían lo posible por utilizar la débil modularidad disponible a través de clases, objetos y cierres. La modularidad basada en cierres, con el apoyo de herramientas de agrupación de código, condujo a una forma práctica de modularidad basada en una función `require()`, que fue adoptada por Node. Los módulos basados en `require()` son una parte fundamental del entorno de programación de Node, pero nunca fueron adoptados como parte oficial del lenguaje JavaScript. En su lugar, ES6 define los módulos mediante las palabras clave `import` y `export`. Aunque la importación y la exportación forman parte del lenguaje desde hace años, los navegadores web y Node las han implementado hace

relativamente poco. Y, en la práctica, la modularidad de JavaScript sigue dependiendo de las herramientas de agrupación de código.

Las secciones que siguen abarcan:

- Módulos "hágalo usted mismo" con clases, objetos y cierres
- Módulos de nodo que utilizan require()
- Módulos ES6 que utilizan export, import e import()

## 10.1 Módulos con clases, objetos y cierres

Aunque sea obvio, vale la pena señalar que una de las características importantes de las clases es que actúan como módulos para sus métodos. Piense en [el Ejemplo 9-8](#). Ese ejemplo definía un número de clases diferentes, todas las cuales tenían un método llamado has(). Pero no tendrías ningún problema en escribir un programa que utilizara varias clases de conjuntos de ese ejemplo: no hay peligro de que la implementación de has() de SingletonSet sobrescriba el método has() de BitSet, por ejemplo.

La razón por la que los métodos de una clase son independientes de los métodos de otras clases no relacionadas es que los métodos de cada clase se definen como propiedades de objetos prototipo independientes. La razón por la que las clases son modulares es que los objetos son modulares: definir una propiedad en un objeto de JavaScript es muy parecido a declarar una variable, pero añadir propiedades a los objetos no afecta al espacio de nombres global de un programa, ni tampoco a las propiedades de otros objetos. JavaScript define bastantes funciones y constantes matemáticas, pero en lugar de definirlas todas globalmente, se agrupan como

propiedades de un único objeto Math global. Esta misma técnica podría haberse utilizado en [el Ejemplo 9-8](#). En lugar de definir clases globales con nombres como SingletonSet y BitSet, ese ejemplo podría haberse escrito para definir sólo un único objeto global Sets, con propiedades que hicieran referencia a las distintas clases. Los usuarios de este

La biblioteca Sets podría entonces referirse a las clases con nombres como Sets.Singleton y Sets.Bit.

El uso de clases y objetos para la modularidad es una técnica común y útil en la programación de JavaScript, pero no va lo suficientemente lejos. En particular, no nos ofrece ninguna forma de ocultar los detalles de la implementación interna dentro del módulo. Consideremos de nuevo [el ejemplo 9-8](#). Si estuviéramos escribiendo ese ejemplo como un módulo, tal vez hubiéramos querido mantener las diversas clases abstractas dentro del módulo, haciendo que las subclases concretas sólo estuvieran disponibles para los usuarios del módulo. De forma similar, en la clase BitSet, los métodos \_valid() y \_has() son utilidades internas que no deberían ser expuestas a los usuarios de la clase. Y BitSet.bits y BitSet.masks son detalles de implementación que sería mejor ocultar.

Como vimos en [§8.6](#), las variables locales y las funciones anidadas declaradas dentro de una función son privadas para esa función. Esto significa que podemos utilizar expresiones de funciones invocadas inmediatamente para lograr una especie de modularidad dejando los detalles de implementación y las funciones de utilidad ocultas dentro de la función adjunta, pero

haciendo que la API pública del módulo sea el valor de retorno de la función. En el caso de la clase BitSet, podríamos estructurar el módulo así:

```
const BitSet = (function() { // Establece BitSet al valor de retorno de esta función //
  Detalles de implementación privados aquí
  function isValid(set, n) { ... }
  function has(set, byte, bit) { ... }
  const BITS = new Uint8Array([1, 2, 4, 8, 16, 32, 64,
    128]);
  const MASKS = new Uint8Array([~1, ~2, ~4, ~8, ~16, ~32, ~64, ~128]);

  // La API pública del módulo es sólo la clase BitSet, que definimos
  // y retornar aquí. La clase puede utilizar las funciones y constantes privadas
  // definidos arriba, pero estarán ocultos para los usuarios de la clase
  return class
    BitSet extends AbstractWritableSet { // ... implementación omitida ...
    };
}());
```

Este enfoque de la modularidad se vuelve un poco más interesante cuando el módulo tiene más de un elemento en él. El siguiente código, por ejemplo, define un minimódulo de estadísticas que exporta las funciones mean() y stddev() dejando ocultos los detalles de implementación:

```

// Así es como podríamos definir un módulo de estadísticas
const stats = function() { // Funciones de utilidad privadas del módulo
  const sum = (x, y) => x + y; const square = x => x * x;

  // Una función pública que será exportada
  function mean(data)
  { return data. reduce(sum)/data. length; }

  // Una función pública que exportaremos
  stddev(data) { let m = mean(data); return Math. sqrt( data.
    map(x => x -
    m). map(square). reduce(sum)/(data. length-1) );
  }

  // Exportamos la función pública como propiedades de un
  objeto
  return { media, stddev }; }());
}

// Y así es como podríamos usar el módulo stats.
mean([1, 3, 5, 7, 9]) // => 5
stats. stddev([1, 3, 5, 7, 9]) // => Math.sqrt(10)

```

### 10.1.1 Automatización de la modularidad basada en el cierre

Tenga en cuenta que es un proceso bastante mecánico para transformar un archivo de código JavaScript en este tipo de módulo mediante la inserción de algún texto al principio y al final del archivo. Todo lo que se necesita es alguna convención para que el archivo de código JavaScript indique qué valores se van a exportar y cuáles no.

Imagine una herramienta que toma un conjunto de archivos, envuelve el contenido de cada uno de esos archivos dentro de una

expresión de función invocada inmediatamente, mantiene un registro del valor de retorno de cada función y concatena todo en un gran archivo. El resultado podría ser algo así:

```
const modules = {};
function require(moduleName) {
    return modules[moduleName];
}

modules["sets.js"] = (function() {
    const exports = {};

    // El contenido del archivo sets.js va aquí: exports.BitSet = class BitSet { ... };

    return exports;
}());

modules["stats.js"] = (function() {
    const exports = {};
    // El contenido del archivo stats.js va aquí:

    const sum = (x, y) => x + y;
    const square = x => x * x;
    exports.mean = function(data) { ... };
    exports.stddev = function(data) { ... };

    return exports;
}());
```

Con los módulos agrupados en un solo archivo como el que se muestra en el ejemplo anterior, puede imaginarse escribir código como el siguiente para hacer uso de esos módulos:

```
// Obtener referencias a los módulos (o al contenido del módulo) que necesitamos
const stats = require("stats.js");
const BitSet = require("sets.js").BitSet;

// Ahora escriba el código usando esos módulos
let s = new BitSet(100);
s.insertar(10);
s.insertar(20);
s.insert(30);
let average = stats.mean([...s]); // La media es 20
```

Este código es un esbozo de cómo funcionan las herramientas de agrupación de código (como webpack y Parcel) para los navegadores web, y también es una simple introducción a la función require() como la que se utiliza en los programas Node.

## 10.2 Módulos en el nodo

En la programación Node, es normal dividir los programas en tantos archivos como parezca natural. Se supone que todos estos archivos de código JavaScript viven en un sistema de archivos rápido. A diferencia de los navegadores web, que tienen que leer los archivos de JavaScript a través de una conexión de red relativamente lenta, no hay necesidad o beneficio de agrupar un programa Node en un solo archivo JavaScript.

En Node, cada archivo es un módulo independiente con un espacio de nombres privado. Las constantes, variables, funciones y clases definidas en un archivo son privadas para ese archivo a menos que éste las exporte. Y los valores exportados por un módulo sólo son visibles en otro módulo si éste los importa explícitamente.

Los módulos de Node importan otros módulos con la función require() y exportan su API pública estableciendo propiedades del objeto Exports o sustituyendo el objeto module.exports por completo.

### 10.2.1 Exportación de nodos

Node define un objeto global de exportación que siempre está definido. Si estás escribiendo un módulo Node que exporta

múltiples valores, puedes simplemente asignarlos a las propiedades de este objeto:

```
const suma = (x, y) => x + y; const cuadrado = x
=> x * x;

exports.mean = data => data.reduce(sum)/data.length; exports.stddev =
function(d) { let m = exports.mean(d); return Math.sqrt(d.map(x => x -
m).map(square).reduce(sum)/(d.length-1));
};
```

Sin embargo, a menudo se quiere definir un módulo que exporte una sola función o clase en lugar de un objeto lleno de funciones o clases. Para ello, basta con asignar a module.exports el único valor que se desea exportar:

```
module.exports = class BitSet extends AbstractWritableSet { // implementación omitida
};
```

El valor por defecto de module.exports es el mismo objeto al que se refiere exports. En el módulo de estadísticas anterior, podríamos haber asignado la función media a module.exports.mean en lugar de a exports.mean. Otro enfoque con módulos como el de estadísticas es exportar un único objeto al final del módulo en lugar de exportar las funciones una a una a medida que se avanza:

```
// Definir todas las funciones, públicas y privadas const sum = (x, y) => x + y; const square = x => x * x; const mean = data => data.reduce(sum)/data.length; const stddev = d => { let m = mean(d); return Math.sqrt(d.map(x => x - m).map(square).reduce(sum)/(d.length-1)); };

// Ahora exporta sólo el módulo de los públicos. exports = { mean, stddev };
```

## 10.2.2 Importación de nodos

Un módulo Node importa otro módulo llamando a la función `require()`. El argumento de esta función es el nombre del módulo que se va a importar, y el valor de retorno es cualquier valor (normalmente una función, clase u objeto) que exporta el módulo.

Si quieras importar un módulo del sistema incorporado a Node o un módulo que hayas instalado en tu sistema a través de un gestor de paquetes, simplemente utiliza el nombre no cualificado del módulo, sin ningún carácter "/" que lo convierta en una ruta del sistema de archivos:

```
// Estos módulos están integrados en Node const fs = require("fs"); // El módulo de sistema de archivos integrado const http = require("http"); // El módulo HTTP integrado

// El marco del servidor HTTP Express es un módulo de terceros.
// No es parte de Node pero se ha instalado localmente const express =
require("express");
```

Cuando quiera importar un módulo de su propio código, el nombre del módulo debe ser la ruta del archivo que contiene ese código, relativa al archivo del módulo actual. Es legal utilizar rutas absolutas que comiencen con un carácter /, pero normalmente, cuando se importan módulos que forman parte de su propio

programa, los nombres de los módulos comenzarán con ./ o a veces ../ para indicar que son relativos al directorio actual o al directorio padre. Por ejemplo:

```
const stats = require('./stats.js'); const BitSet =  
require('./utils/bitset.js');
```

(También puedes omitir el sufijo .js en los archivos que estás importando y Node seguirá encontrando los archivos, pero es común ver estas extensiones de archivo explícitamente incluidas).

Cuando un módulo exporta una sola función o clase, todo lo que tienes que hacer es requerirla. Cuando un módulo exporta un objeto con múltiples propiedades, tienes una opción: puedes importar todo el objeto, o sólo importar las propiedades específicas (usando la asignación de desestructuración) del objeto que planeas usar. Compare estos dos enfoques:

```
// Importar el objeto stats completo, con todas sus funciones  
const stats =  
require('./stats.js');

// Tenemos más funciones de las que necesitamos, pero están ordenadas  
// organizado en un conveniente espacio de nombres "stats". let average =  
stats.mean(data);

// Alternativamente, podemos utilizar la asignación idiomática de  
desestructuración para importar // exactamente las funciones que queremos  
directamente en el espacio de nombres local: const { stddev } =  
require('./stats.js');

// Esto es bonito y sucinto, aunque perdemos un poco de contexto // sin el prefijo 'stats'  
// como namespace para la función stddev().  
let sd = stddev(data);
```

### 10.2.3 Módulos tipo nodo en la web

Los módulos con un objeto Exports y una función require() están incorporados en Node. Pero si estás dispuesto a procesar tu código

con una herramienta de empaquetamiento como webpack, entonces también es posible utilizar este estilo de módulos para el código que está destinado a ejecutarse en los navegadores web. Hasta hace poco, esto era algo muy común, y es posible que veas mucho código basado en la web que todavía lo hace.

Sin embargo, ahora que JavaScript tiene su propia sintaxis de módulo estándar, los desarrolladores que utilizan bundlers son más propensos a utilizar los módulos oficiales de JavaScript con declaraciones import y export.

## 10.3 Módulos en ES6

ES6 añade las palabras clave "import" y "export" a JavaScript y, por fin, admite la modularidad real como característica principal del lenguaje. La modularidad de ES6 es conceptualmente la misma que la de Node: cada archivo es su propio módulo, y las constantes, variables, funciones y clases definidas dentro de un archivo son privadas para ese módulo a menos que se exporten explícitamente. Los valores que se exportan de un módulo están disponibles para su uso en los módulos que los importan explícitamente. Los módulos ES6 difieren de los módulos Node en la sintaxis utilizada para exportar e importar y también en la forma en que se definen los módulos en los navegadores web. Las secciones que siguen explican estas cosas en detalle.

En primer lugar, sin embargo, hay que tener en cuenta que los módulos de ES6 también se diferencian de los "scripts" normales de JavaScript en algunos aspectos importantes. La diferencia más obvia es la propia modularidad: en los scripts normales, las

declaraciones de nivel superior de variables, funciones y clases van a un único contexto global compartido por todos los scripts. Con los módulos, cada archivo tiene su propio contexto privado y puede utilizar las sentencias import y export, que es de lo que se trata, después de todo. Pero también hay otras diferencias entre los módulos y los scripts. El código dentro de un módulo ES6 (como el código dentro de cualquier definición de clase ES6) está automáticamente en modo estricto (véase §5.6.3). Esto significa que, cuando empiece a usar módulos ES6, no tendrá que volver a escribir "use strict". Y significa que el código en los módulos no puede usar la sentencia with o el objeto arguments o variables no declaradas. Los módulos ES6 son incluso un poco más estrictos que el modo estricto: en el modo estricto, en las funciones invocadas como funciones, esto es indefinido. En los módulos, esto es indefinido incluso en el código de nivel superior. (Por el contrario, los scripts en los navegadores web y en Node establecen esto en el objeto global).

## MÓDULOS ES6 EN LA WEB Y EN NODE

Los módulos ES6 llevan años utilizándose en la web con la ayuda de agrupadores de código como webpack, que combinan módulos independientes de código JavaScript en grandes paquetes no modulares adecuados para su inclusión en las páginas web. Sin embargo, en el momento de escribir este artículo, los módulos ES6 son finalmente soportados de forma nativa por todos los navegadores web excepto Internet Explorer. Cuando se utilizan de forma nativa, los módulos ES6 se añaden a las páginas HTML con una etiqueta especial `<script type="module">`, que se describe más adelante en este capítulo.

Y mientras tanto, habiendo sido pionero en la modularidad de JavaScript, Node se encuentra en la incómoda posición de tener que soportar dos sistemas de módulos no del todo compatibles. Node 13 es compatible con los módulos ES6, pero por ahora, la gran mayoría de los programas Node siguen utilizando módulos Node.

### 10.3.1 Exportación de ES6

Para exportar una constante, una variable, una función o una clase de un módulo ES6, basta con añadir la palabra clave `export` antes de la declaración:

```
export const PI = Math.PI; export function degreesToRadians(d) { return d * PI / 180;

}

export class Círculo { constructor(r) { this.r = r; } area() {
  return PI * this.r * this.r;
}
```

Como alternativa a la dispersión de las palabras clave de exportación a lo largo de su módulo, puede definir sus constantes, variables, funciones y clases como lo haría normalmente, sin declaración de exportación, y luego (normalmente al final de su módulo) escribir una sola declaración de exportación que declare exactamente lo que se exporta en un solo lugar. Así, en lugar de escribir tres exportaciones individuales en el código anterior, podríamos haber escrito de forma equivalente una sola línea al final:

```
exportar { Circle, degreesToRadians, PI };
```

Esta sintaxis se parece a la palabra clave `export` seguida de un literal de objeto (utilizando la notación abreviada). Pero en este caso, las llaves no definen realmente un literal de objeto. Esta sintaxis de exportación simplemente requiere una lista de identificadores separados por comas entre llaves.

Es común escribir módulos que exportan sólo un valor (típicamente una función o clase), y en este caso, solemos utilizar `export default` en lugar de `export`:

```
export default class BitSet { // implementación omitida  
}
```

Las exportaciones por defecto son un poco más fáciles de importar que las exportaciones no por defecto, así que cuando sólo hay un valor exportado, usar la exportación por defecto facilita las cosas para los módulos que usan su valor exportado.

Las exportaciones regulares con `export` sólo pueden realizarse en declaraciones que tengan un nombre. Las exportaciones por defecto con `export default` pueden exportar cualquier expresión, incluyendo expresiones de funciones anónimas y expresiones de clases anónimas. Esto significa que si usas `export default`, puedes exportar literales de objetos. Así que, a diferencia de la sintaxis de exportación, si ves llaves después de `export default`, realmente es un literal de objeto lo que se está exportando.

Es legal, aunque poco común, que los módulos tengan un conjunto de exportaciones regulares y también una exportación por defecto. Si un módulo tiene una exportación por defecto, sólo puede tener una.

Por último, tenga en cuenta que la palabra clave `export` sólo puede aparecer en el nivel superior de su código JavaScript. No se puede exportar un valor desde dentro de una clase, función, bucle o condicional. (Esta es una característica importante del sistema de módulos de ES6 y permite el análisis estático: la exportación de un módulo será la misma en cada ejecución, y los símbolos exportados pueden determinarse antes de que el módulo se ejecute realmente).

### 10.3.2 Importaciones de ES6

Con la palabra clave import se importan valores que han sido exportados por otros módulos. La forma más sencilla de importar se utiliza para los módulos que definen una exportación por defecto:

```
import BitSet from './bitset.js';
```

Se trata de la palabra clave import, seguida de un identificador, seguida de la palabra clave from, seguida de una cadena literal que nombra el módulo cuya exportación por defecto estamos importando. El valor de exportación por defecto del módulo especificado se convierte en el valor del identificador especificado en el módulo actual.

El identificador al que se asigna el valor importado es una constante, como si se hubiera declarado con la palabra clave const. Al igual que las exportaciones, las importaciones sólo pueden aparecer en el nivel superior de un módulo y no están permitidas dentro de clases, funciones, bucles o condicionales. Por convención casi universal, las importaciones que necesita un módulo se colocan al principio del mismo. Sin embargo, esto no es necesario: al igual que las declaraciones de funciones, las importaciones se "elevan" a la parte superior, y todos los valores importados están disponibles para cualquier ejecución de código del módulo.

El módulo del que se importa un valor se especifica como un literal de cadena constante entre comillas simples o dobles. (No puede utilizar una variable u otra expresión cuyo valor sea una cadena, y

no puede utilizar una cadena entre comillas porque los literales de plantilla pueden interpolar variables y no siempre tienen valores constantes). En los navegadores web, esta cadena se interpreta como una URL relativa a la ubicación del módulo que está haciendo la importación. (En Node, o cuando se utiliza una herramienta de empaquetado, la cadena se interpreta como un nombre de archivo relativo al módulo actual, pero esto supone poca diferencia en la práctica). Una cadena de *especificación de módulo* debe ser una ruta absoluta que empiece por "/", o una ruta relativa que empiece por "./" o "../", o una URL completa con protocolo y nombre de host. La especificación ES6 no permite cadenas especificadoras de módulos no cualificadas como "util.js" porque es ambiguo si esto pretende nombrar un módulo en el mismo directorio que el actual o algún tipo de módulo del sistema que está instalado en alguna ubicación especial. (Esta restricción contra los "especificadores de módulos desnudos" no es respetada por las herramientas de agrupación de código como webpack, que pueden configurarse fácilmente para encontrar módulos desnudos en un directorio de biblioteca que se especifique). Una futura versión del lenguaje puede permitir "especificadores de módulos desnudos", pero por ahora no están permitidos. Si quieres importar un módulo del mismo directorio que el actual, simplemente coloca "./" antes del nombre del módulo e importa de "./util.js" en lugar de "util.js".

Hasta ahora, sólo hemos considerado el caso de importar un único valor de un módulo que utiliza exportar por defecto. Para importar valores de un módulo que exporta múltiples valores, utilizamos una sintaxis ligeramente diferente:

```
import { mean, stddev } from "./stats.js";
```

Recordemos que las exportaciones por defecto no necesitan tener un nombre en el módulo que las define. En su lugar, proporcionamos un nombre local cuando importamos esos valores. Pero las exportaciones no predeterminadas de un módulo sí tienen nombres en el módulo exportador, y cuando importamos esos valores, nos referimos a ellos por esos nombres. El módulo exportador puede exportar cualquier número de valores con nombre. Una sentencia import que haga referencia a ese módulo puede importar cualquier subconjunto de esos valores simplemente enumerando sus nombres entre llaves. Las llaves hacen que este tipo de declaración de importación se parezca a una asignación de desestructuración, y la asignación de desestructuración es en realidad una buena analogía para lo que hace este estilo de importación. Los identificadores dentro de las llaves se elevan a la parte superior del módulo de importación y se comportan como constantes.

Las guías de estilo a veces recomiendan que se importen explícitamente todos los símbolos que utilizará el módulo. Sin embargo, cuando se importa desde un módulo que define muchas exportaciones, se puede importar todo fácilmente con una sentencia import como esta:

```
import * as stats from "./stats.js";
```

Una declaración de importación como ésta crea un objeto y lo asigna a una constante llamada stats. Cada una de las exportaciones no predeterminadas del módulo que se importa se convierte en una propiedad de este objeto stats. Las exportaciones

no predeterminadas siempre tienen nombres, y éstos se utilizan como nombres de propiedades dentro del objeto. Estas propiedades son efectivamente constantes: no pueden ser sobrescritas o borradas. Con la importación comodín mostrada en el ejemplo anterior, el módulo importador utilizaría las funciones importadas `mean()` y `stddev()` a través del objeto `stats`, invocándolas como `stats.mean()` y `stats.stddev()`.

Los módulos suelen definir una exportación por defecto o varias exportaciones con nombre. Es legal, aunque poco común, que un módulo utilice tanto la exportación como la exportación por defecto. Pero cuando un módulo hace eso, puedes importar tanto el valor por defecto como los valores con nombre con una sentencia `import` como esta:

```
import Histogram, { mean, stddev } from "./histogramstats.js";
```

Hasta ahora, hemos visto cómo importar desde módulos con una exportación por defecto y desde módulos con exportaciones no por defecto o con nombre. Pero hay otra forma de la sentencia `import` que se utiliza con módulos que no tienen ninguna exportación. Para incluir un módulo sin exportaciones en su programa, simplemente utilice la palabra clave `import` con el especificador de módulo:

```
importar "./analytics.js";
```

Un módulo como éste se ejecuta la primera vez que se importa. (Y las importaciones posteriores no hacen nada.) Un módulo que sólo define funciones sólo es útil si exporta al menos una de esas funciones. Pero si un módulo ejecuta algún código, entonces

puede ser útil importarlo incluso sin símbolos. Un módulo de análisis para una aplicación web podría ejecutar código para registrar varios manejadores de eventos y luego usar esos manejadores de eventos para enviar datos de telemetría al servidor en los momentos adecuados. El módulo es autónomo y no necesita exportar nada, pero todavía necesitamos para importarlo y que realmente se ejecute como parte de nuestro programa.

Tenga en cuenta que puede utilizar esta sintaxis de importación de nada incluso con módulos que tienen exportaciones. Si un módulo define un comportamiento útil independiente de los valores que exporta, y si su programa no necesita ninguno de esos valores exportados, puede seguir importando el módulo... sólo para ese comportamiento por defecto.

### 10.3.3 Importaciones y exportaciones con cambio de nombre

Si dos módulos exportan dos valores diferentes con el mismo nombre y usted quiere importar ambos valores, tendrá que renombrar uno o ambos valores cuando los importe. Del mismo modo, si quieres importar un valor cuyo nombre ya está en uso en tu módulo, tendrás que renombrar el valor importado. Puedes utilizar la palabra clave `as` con las importaciones con nombre para renombrarlas mientras las importas:

```
import { render como renderImage } de "./imageutils.js"; import { render como renderUI } de "./ui.js";
```

Estas líneas importan dos funciones al módulo actual. Ambas funciones se llaman `render()` en los módulos que las definen pero se importan con los nombres más descriptivos y desambiguadores `renderImage()` y `renderUI()`.

Recordemos que las exportaciones por defecto no tienen nombre. El módulo importador siempre elige el nombre cuando importa una exportación por defecto. Así que no hay necesidad de una sintaxis especial para renombrar en ese caso.

Dicho esto, sin embargo, la posibilidad de renombrar en la importación proporciona otra forma de importar desde módulos que definen tanto una exportación por defecto como exportaciones con nombre. Recuerda el módulo `"./histogram-stats.js"` de la sección anterior. Esta es otra forma de importar tanto las exportaciones por defecto como las exportaciones con nombre de ese módulo:

```
import { default as Histogram, mean, stddev } from "./histogram-stats.js";
```

En este caso, la palabra clave JavaScript `default` sirve como marcador de posición y nos permite indicar que queremos importar y proporcionar un nombre para la exportación por defecto del módulo.

También es posible renombrar los valores a medida que se exportan, pero sólo cuando se utiliza la variante de la sentencia `export` con llaves. No es común necesitar hacer esto, pero si eligió nombres cortos y sucintos para usar dentro de su módulo, podría preferir exportar sus valores con nombres más descriptivos que sean menos propensos a entrar en conflicto con otros módulos. Al

igual que con las importaciones, se utiliza la palabra clave `as` para hacer esto:

```
export { layout como calculateLayout,  
render como renderLayout };
```

Tenga en cuenta que, aunque las llaves parecen algo así como literales de objeto, no lo son, y la palabra clave `export` espera un único identificador antes del `as`, no una expresión. Esto significa, desafortunadamente, que no puede utilizar el renombramiento de exportación de esta manera:

```
export { Math.sin como sin, Math.cos como cos }; // SyntaxError
```

### 10.3.4 Reexportaciones

A lo largo de este capítulo, hemos discutido un módulo hipotético `./stats.js` que exporta las funciones `mean()` y `stddev()`. Si estuviéramos escribiendo un módulo de este tipo y pensáramos que muchos usuarios del módulo querrían sólo una función o la otra, entonces podríamos definir `mean()` en un módulo `./stats/mean.js` y definir `stddev()` en `./stats/stddev.js`. De este modo, los programas sólo tienen que importar exactamente las funciones que necesitan y no se hinchan importando código que no necesitan.

Sin embargo, incluso si hubiéramos definido estas funciones estadísticas en módulos individuales, podríamos esperar que hubiera muchos programas que quisieran ambas funciones y agradecerían un práctico módulo `./stats.js` desde el que pudieran importar ambas en una sola línea.

Dado que las implementaciones están ahora en archivos separados, definir este módulo "./stat.js" es sencillo:

```
import { mean } from "./stats/mean.js"; import { stddev } from  
"./stats/stddev.js"; export { mean, stddev };
```

Los módulos ES6 anticipan este caso de uso y proporcionan una sintaxis especial para ello. En lugar de importar un símbolo simplemente para exportarlo de nuevo, puedes combinar los pasos de importación y exportación en una única sentencia "re-export" que utiliza la palabra clave `export` y la palabra clave `from`:

```
export { mean } from "./stats/mean.js"; export { stddev } from  
"./stats/stddev.js";
```

Tenga en cuenta que los nombres `media` y `stddev` no se utilizan realmente en este código. Si no estamos siendo selectivos con una reexportación y simplemente queremos exportar todos los valores con nombre de otro módulo, podemos utilizar un comodín:

```
export * from "./stats/mean.js"; export * from  
"./stats/stddev.js";
```

La sintaxis de reexportación permite renombrar con como lo hacen las sentencias regulares de importación y exportación.

Supongamos que queremos reexportar la función `mean()` pero también definir `average()` como otro nombre para la función.

Podríamos hacerlo así:

```
export { mean, mean as average } from "./stats/mean.js"; export { stddev } from  
"./stats/stddev.js";
```

Todas las reexportaciones de este ejemplo suponen que los módulos `./stats/mean.js` y `./stats/stddev.js` exportan sus funciones utilizando `export` en lugar de `export default`. Sin

embargo, como se trata de módulos con una sola exportación, habría sentido definirlos con `export default`. Si lo hubiéramos hecho así, la sintaxis de reexportación es un poco más complicada porque necesita definir un nombre para las exportaciones por defecto sin nombre. Podemos hacerlo así:

```
export { default as mean } from "./stats/mean.js"; export { default as stddev }  
from "./stats/stddev.js";
```

Si quieras reexportar un símbolo con nombre de otro módulo como exportación por defecto de tu módulo, podrías hacer una importación seguida de una exportación por defecto, o podrías combinar las dos sentencias así:

```
// Importar la función mean() de ./stats.js y convertirla en la  
// exportación por defecto de este módulo export { mean as  
default } from "./stats.js"
```

Y finalmente, para reexportar la exportación por defecto de otro módulo como la exportación por defecto de su módulo (aunque no está claro por qué querría hacer esto, ya que los usuarios podrían simplemente importar el otro módulo directamente), puede escribir:

```
// El módulo average.js simplemente reexporta la exportación por defecto de  
stats/mean.js  
export { default } from "./stats/mean.js"
```

### 10.3.5 Módulos JavaScript en la Web

Las secciones anteriores han descrito los módulos de ES6 y sus declaraciones de importación y exportación de una manera algo abstracta. En esta sección y en la siguiente, discutiremos cómo funcionan realmente en los navegadores web, y si no eres ya un desarrollador web experimentado, puede que encuentres el resto

de este capítulo más fácil de entender después de haber leído [el Capítulo 15](#).

A principios de 2020, el código de producción que utiliza módulos ES6 todavía se agrupa generalmente con una herramienta como webpack. Esto tiene sus [contrapartidas<sup>1</sup>](#), pero en general, la agrupación del código tiende a ofrecer un mejor rendimiento. Esto podría cambiar en el futuro, a medida que la velocidad de la red aumente y los proveedores de navegadores sigan optimizando sus implementaciones de módulos ES6.

Aunque las herramientas de agrupación pueden seguir siendo deseables en producción, ya no son necesarias en el desarrollo ya que todos los navegadores actuales proporcionan soporte nativo para los módulos de JavaScript. Recordemos que los módulos utilizan el modo estricto por defecto, éste no se refiere a un objeto global, y las declaraciones de nivel superior no se comparten globalmente por defecto. Dado que los módulos deben ejecutarse de forma diferente al código heredado sin módulos, su introducción requiere cambios tanto en HTML como en JavaScript. Si quieras utilizar de forma nativa las directivas de importación en un navegador web, debes indicarle al navegador web que tu código es un módulo utilizando una etiqueta `<script type="module">`.

Una de las buenas características de los módulos ES6 es que cada módulo tiene un conjunto estático de importaciones. Así que dado un único módulo inicial, un navegador web puede cargar todos sus módulos importados y luego cargar todos los módulos importados por ese primer lote de módulos, y así sucesivamente, hasta que se

haya cargado un programa completo. Hemos visto que el especificador de módulo en una declaración de importación puede ser tratado como una URL relativa. Una etiqueta `<script type="module">` marca el punto de partida de un programa modular. Sin embargo, no se espera que ninguno de los módulos que importa esté en etiquetas `<script>`: en su lugar, se cargan bajo demanda como archivos JavaScript normales y se ejecutan en modo estricto como módulos ES6 normales. El uso de una etiqueta `<script type="module">` para definir el punto de entrada principal de un programa modular de JavaScript puede ser tan simple como esto:

```
<script type="module"> importar "./main.js"; </script>
```

El código dentro de una etiqueta inline `<script type="module">` es un módulo ES6, y como tal puede utilizar la sentencia `export`. Sin embargo, no tiene sentido hacerlo porque la sintaxis de la etiqueta HTML `<script>` no proporciona ninguna forma de definir un nombre para los módulos en línea, por lo que incluso si dicho módulo exporta un valor, no hay forma de que otro módulo lo importe.

Los scripts con el atributo `type="module"` se cargan y ejecutan como los scripts con el atributo `defer`. La carga del código comienza en cuanto el analizador HTML encuentra la etiqueta `<script>` (en el caso de los módulos, este paso de carga de código puede ser un proceso recursivo que cargue varios archivos JavaScript). Pero la ejecución del código no comienza hasta que se completa el análisis sintáctico de HTML. Y una vez completado el análisis sintáctico del HTML, los scripts (tanto modulares como no

modulares) se ejecutan en el orden en que aparecen en el documento HTML.

Puedes modificar el tiempo de ejecución de los módulos con el atributo `async`, que funciona de la misma manera para los módulos que para los scripts regulares. Un módulo asíncrono se ejecutará tan pronto como se cargue el código, incluso si el análisis sintáctico de HTML no se ha completado e incluso si esto cambia el orden relativo de los scripts.

Los navegadores web que soportan `<script type="module">` también deben soportar `<script nomodule>`. Los navegadores que admiten módulos ignoran cualquier script con el atributo `nomodule` y no lo ejecutan. Los navegadores que no soportan módulos no reconocerán el atributo `nomodule`, por lo que lo ignorarán y ejecutarán el script. Esto proporciona una poderosa técnica para lidiar con los problemas de compatibilidad de los navegadores. Los navegadores que soportan módulos ES6 también soportan otras características modernas de JavaScript como clases, funciones de flecha y el bucle `for/of`.

Si escribes JavaScript moderno y lo cargas con `<script type="module">`, sabes que sólo lo cargarán los navegadores que lo soporten. Y como alternativa para IE11 (que, en 2020, es efectivamente el único navegador que no soporta ES6), puedes utilizar herramientas como Babel y webpack para transformar tu código en código ES5 no modular, y luego cargar ese código transformado menos eficiente a través de `<script nomodule>`.

Otra diferencia importante entre los scripts normales y los scripts de módulo tiene que ver con la carga entre orígenes. Una etiqueta `<script>` normal cargará un archivo de código JavaScript desde cualquier servidor de Internet, y la infraestructura de publicidad, análisis y código de seguimiento de Internet depende de este hecho. Pero `<script type="module">` ofrece la oportunidad de endurecer esto, y los módulos sólo pueden cargarse desde el mismo origen que el documento HTML que los contiene o cuando las cabeceras CORS adecuadas están en su lugar para permitir de forma segura las cargas entre orígenes. Un desafortunado efecto secundario de esta nueva restricción de seguridad es que dificulta la prueba de los módulos ES6 en modo de desarrollo utilizando archivos: URL. Cuando se utilicen módulos ES6, es probable que sea necesario configurar un servidor web estático para las pruebas.

A algunos programadores les gusta utilizar la extensión de nombre de archivo `.mjs` para distinguir sus archivos JavaScript modulares de sus archivos JavaScript normales, no modulares, con la extensión tradicional `.js`. A efectos de los navegadores web y las etiquetas `<script>`, la extensión del archivo es en realidad irrelevante. (Sin embargo, el tipo MIME sí es relevante, por lo que si utiliza archivos `.mjs`, puede que tenga que configurar su servidor web para que los sirva con el mismo tipo MIME que los archivos `.js`). El soporte de Node para ES6 utiliza la extensión del nombre del archivo como una pista para distinguir qué sistema de módulos utiliza cada archivo que carga. Así que si estás escribiendo módulos ES6 y quieres que sean utilizables con Node, entonces puede ser útil adoptar la convención de nomenclatura `.mjs`.

### 10.3.6 Importaciones dinámicas con import()

Hemos visto que las directivas de importación y exportación de ES6 son completamente estáticas y permiten a los intérpretes de JavaScript y a otras herramientas de JavaScript determinar las relaciones entre los módulos con un simple análisis de texto mientras se cargan los módulos sin tener que ejecutar realmente ningún código de los módulos. Con los módulos importados estáticamente, se garantiza que los valores que se importan a un módulo estarán listos para su uso antes de que comience a ejecutarse el código del módulo.

En la web, el código tiene que ser transferido a través de una red en lugar de ser leído desde el sistema de archivos. Y una vez transferido, ese código suele ejecutarse en dispositivos móviles con CPUs relativamente lentes. Este no es el tipo de entorno en el que las importaciones de módulos estáticos -que requieren que se cargue todo un programa antes de que se ejecute- tienen mucho sentido.

Es común que las aplicaciones web carguen inicialmente sólo lo suficiente de su código para renderizar la primera página mostrada al usuario. Entonces, una vez que el usuario tiene algún contenido preliminar con el que interactuar, puede empezar a cargar la cantidad de código, a menudo mucho mayor, que se necesita para el resto de la aplicación web. Los navegadores web facilitan la carga dinámica de código mediante el uso de la API DOM para injectar una nueva etiqueta <script> en el documento HTML actual, y las aplicaciones web han estado haciendo esto durante muchos años.

Aunque la carga dinámica ha sido posible durante mucho tiempo, no ha formado parte del lenguaje en sí. Eso cambia con la introducción de `import()` en ES2020 (a partir de principios de 2020, la importación dinámica es soportada por todos los navegadores que soportan módulos ES6). Se pasa un especificador de módulo a `import()` y éste devuelve un objeto Promise que representa el proceso asíncrono de carga y ejecución del módulo especificado.

Cuando la importación dinámica se completa, la promesa se "cumple" (véase

[Capítulo 13](#) para obtener detalles completos sobre la programación asíncrona y las promesas) y produce un objeto como el que se obtendría con la forma `import *` de la sentencia `static import`.

Así que en lugar de importar el módulo `./stats.js` estáticamente, así

```
import * as stats from "./stats.js";
```

podríamos importarlo y utilizarlo dinámicamente, así:

```
import("./stats.js").then(stats => { let average =  
  stats.mean(data); })
```

O, en una función asíncrona (de nuevo, puede que necesites leer [el capítulo 13](#) antes de entender este código), podemos simplificar el código con `await`:

```
async analyzeData(data) { let stats = await import("./stats.js"); return {  
  average: stats.mean(data), stddev: stats.stddev(data) };  
}
```

El argumento de `import()` debe ser un especificador de módulo, exactamente igual que el que se usaría con una directiva de

importación estática. Pero con import(), no está obligado a utilizar una cadena literal constante: cualquier expresión que se evalúe a una cadena en la forma adecuada servirá.

Dynamic import() parece una invocación a una función, pero en realidad no lo es. En su lugar, import() es un operador y los paréntesis son una parte necesaria de la sintaxis del operador. La razón de esta inusual sintaxis es que import() necesita ser capaz de resolver los especificadores de módulo como URLs relativas al módulo que se está ejecutando, y esto requiere un poco de magia de implementación que no sería legal poner en una función de JavaScript. La distinción entre función y operador rara vez supone una diferencia en la práctica, pero lo notarás si intentas escribir código como console.log(import); o let require = import;.

Por último, tenga en cuenta que dynamic import() no es sólo para los navegadores web. Las herramientas de empaquetado de código como webpack también pueden hacer un buen uso de ella. La forma más directa de utilizar un empaquetador de código es indicarle el punto de entrada principal de tu programa y dejar que encuentre todas las directivas de importación estáticas y ensamble todo en un gran archivo. Sin embargo, utilizando estratégicamente las llamadas dinámicas a import(), puedes dividir ese paquete monolítico en un conjunto de paquetes más pequeños que pueden ser cargados bajo demanda.

### **10.3.7 import.meta.url**

Hay una última característica del sistema de módulos de ES6 que hay que discutir. Dentro de un módulo ES6 (pero no dentro de un <script> normal o un módulo Node cargado con require()), la

sintaxis especial `import.meta` se refiere a un objeto que contiene metadatos sobre el módulo que se está ejecutando. La propiedad `url` de este objeto es la URL desde la que se cargó el módulo. (En Node, esto será una URL `file://`).

El principal caso de uso de `import.meta.url` es poder referirse a imágenes, archivos de datos u otros recursos que estén almacenados en el mismo directorio que el módulo (o relativos a él). El constructor `URL()` facilita la resolución de una URL relativa contra una URL absoluta como `import.meta.url`. Suponga, por ejemplo, que ha escrito un módulo que incluye cadenas que necesitan ser localizadas y que los archivos de localización están almacenados en un directorio `I10n/`, que está en el mismo directorio que el propio módulo. Su módulo podría cargar sus cadenas utilizando una URL creada con una función, como esta:

```
function localStringsURL(locale) { return new URL(`I10n/${locale}.json`, import.meta.url); }
```

## 10.4 Resumen

El objetivo de la modularidad es permitir a los programadores ocultar los detalles de implementación de su código, de manera que se puedan ensamblar trozos de código de varias fuentes en programas grandes sin preocuparse de que un trozo sobrescriba funciones o variables de otro. En este capítulo se han explicado tres sistemas de módulos de JavaScript diferentes:

- En los primeros días de JavaScript, la modularidad sólo podía lograrse mediante el uso inteligente de expresiones de función de invocación inmediata.

- Node añadió su propio sistema de módulos sobre el lenguaje JavaScript. Los módulos de Node se importan con `require()` y definen sus exportaciones estableciendo propiedades del objeto `Exports`, o estableciendo la propiedad `module.exports`.
  - En ES6, JavaScript finalmente obtuvo su propio sistema de módulos con palabras clave de importación y exportación, y ES2020 está añadiendo soporte para importaciones dinámicas con `import()`.
- 

Por ejemplo: las aplicaciones web que tienen frecuentes actualizaciones incrementales y los usuarios que hacen

<sup>1</sup> las visitas frecuentes pueden encontrar que el uso de módulos pequeños en lugar de paquetes grandes puede resultar en mejores tiempos de carga promedio debido a una mejor utilización de la caché del navegador del usuario.



# Capítulo 11. La biblioteca estándar de JavaScript

---

Algunos tipos de datos, como los números y las cadenas ([capítulo 3](#)), los objetos ([capítulo 6](#)) y las matrices ([capítulo 7](#)) son tan fundamentales para JavaScript que podemos considerarlos parte del propio lenguaje. Este capítulo cubre otras APIs importantes pero menos fundamentales que pueden considerarse como la definición de la "biblioteca estándar" de JavaScript: son clases y funciones útiles que están incorporadas a JavaScript y disponibles para todos los programas de JavaScript tanto en los navegadores web como en Node.<sup>1</sup>

Las secciones de este capítulo son independientes entre sí, y puede leerlas en cualquier orden. Abarcan:

- Las clases Set y Map para representar conjuntos de valores y mapeos de un conjunto de valores a otro conjunto de valores.
- Objetos tipo array conocidos como TypedArrays que representan arrays de datos binarios, junto con una clase relacionada para extraer valores de datos binarios no array.
- Las expresiones regulares y la clase RegExp, que definen patrones textuales y son útiles para el procesamiento de textos. Esta sección también cubre la sintaxis de las expresiones regulares en detalle.

- La clase Date para representar y manipular fechas y horas.
- La clase Error y sus diversas subclases, cuyas instancias se lanzan cuando se producen errores en los programas de JavaScript.
- El objeto JSON, cuyos métodos admiten la serialización y deserialización de estructuras de datos de JavaScript compuestas por objetos, matrices, cadenas, números y booleanos.
- El objeto Intl y las clases que define que pueden ayudarle a localizar sus programas de JavaScript.
- El objeto Console, cuyos métodos dan salida a las cadenas de manera que son particularmente útiles para depurar programas y registrar el comportamiento de esos programas.
- La clase URL, que simplifica la tarea de analizar y manipular URLs. Esta sección también cubre las funciones globales para codificar y decodificar URLs y sus partes componentes.
- setTimeout() y funciones relacionadas para especificar el código que se ejecutará después de que haya transcurrido un intervalo de tiempo determinado.

Algunas de las secciones de este capítulo -en particular, las secciones sobre arrays tipados y expresiones regulares- son bastante largas porque hay una importante información de fondo que es necesario comprender antes de poder utilizar esos tipos de forma eficaz. Muchas de las otras secciones, sin embargo, son cortas: simplemente introducen una nueva API y muestran algunos ejemplos de su uso.

## 11.1 Conjuntos y mapas

El tipo de objeto de JavaScript es una estructura de datos versátil que puede utilizarse para asignar cadenas (los nombres de las propiedades del objeto) a valores arbitrarios. Y cuando el valor que se asigna es algo fijo como true, entonces el objeto es efectivamente un conjunto de cadenas.

En realidad, los objetos se utilizan como mapas y conjuntos de forma bastante rutinaria en la programación de JavaScript, pero esto está limitado por la restricción a las cadenas y se complica por el hecho de que los objetos normalmente heredan propiedades con nombres como "toString", que no suelen formar parte del mapa o conjunto.

Por esta razón, ES6 introduce las verdaderas clases Set y Map, que cubriremos en las subsecciones siguientes.

### 11.1.1 La clase Set

Un conjunto es una colección de valores, como una matriz. Sin embargo, a diferencia de las matrices, los conjuntos no están ordenados ni indexados, y no admiten duplicados: un valor es miembro de un conjunto o no lo es; no es posible preguntar cuántas veces aparece un valor en un conjunto.

Crea un objeto Set con el constructor Set():

```
let s = new Set(); // Un nuevo conjunto vacío let t = new Set([1, s]); // Un nuevo conjunto con dos miembros
```

El argumento del constructor de Set() no necesita ser un array: se permite cualquier objeto iterable (incluyendo otros objetos Set):

```
let t = new Set(s); // Un nuevo conjunto que copia los elementos de s.  
let unique = new Set("Mississippi"); // 4 elementos: "M", "i", "s" y "p"
```

La propiedad de tamaño de un conjunto es como la propiedad de longitud de un array: indica cuántos valores contiene el conjunto:

```
único. tamaño //=> 4
```

Los conjuntos no necesitan ser inicializados cuando se crean.

Puede añadir y eliminar elementos en cualquier momento con add(), delete() y clear(). Recuerde que los conjuntos no pueden contener duplicados, por lo que añadir un valor a un conjunto cuando ya contiene ese valor no tiene ningún efecto:

```
let s = new Set(); // Comienza vacío  
s. tamaño //=> 0  
s. add(1); // Añadir un número  
s. size //=> 1; ahora el conjunto tiene un miembro  
s. add(1); // Vuelve a añadir el mismo número  
s. tamaño //=> 1; el tamaño no cambia  
s. add(true); // Añade otro valor; ten en cuenta que está bien mezclar tipos  
s. tamaño //=> 2  
s. add([1,2,3]); // Añadir un valor de matriz  
s. size //=> 3; se ha añadido el array, no sus elementos  
s. delete(1) //=> true: se ha eliminado con éxito el elemento 1  
s. tamaño //=> 2: el tamaño vuelve a ser 2  
s. delete("test") //=> false: "test" no era un miembro, el borrado ha fallado  
s. delete(true) //=> true: borrado exitoso  
s. delete([1,2,3]) //=> false: el array en el conjunto es diferente  
s. size //=> 1: todavía existe ese array en el conjunto  
s. clear(); // Eliminar todo del conjunto  
s. tamaño //=> 0
```

Hay algunos puntos importantes a tener en cuenta sobre este código:

- El método `add()` toma un único argumento; si pasas un array, añade el propio array al conjunto, no los elementos individuales del array. Sin embargo, `add()` siempre devuelve el conjunto sobre el que se invoca, por lo que si quieras añadir varios valores a un conjunto, puedes utilizar llamadas a métodos encadenados como

```
s.add('a').add('b').add('c');
```

- El método `delete()` también borra un solo elemento del conjunto a la vez. Sin embargo, a diferencia de `add()`, `delete()` devuelve un valor booleano. Si el valor especificado era realmente un miembro del conjunto, entonces `delete()` lo elimina y devuelve `true`. En caso contrario, no hace nada y devuelve `false`.
- Por último, es muy importante entender que la pertenencia a un conjunto se basa en comprobaciones de igualdad estrictas, como las que realiza el operador `==`. Un conjunto puede contener tanto el número 1 como la cadena "1", porque los considera valores distintos. Cuando los valores son objetos (o arrays o funciones), también se comparan como con `==`. Por eso no pudimos eliminar el elemento del array del conjunto en este código. Añadimos un array al conjunto y luego intentamos eliminar ese array pasando un array *diferente* (aunque con los mismos elementos) al método `delete()`. Para que esto funcione, tendríamos que haber pasado una referencia a exactamente el mismo array.

#### NOTA

Los programadores de Python toman nota: esta es una diferencia significativa entre los conjuntos de JavaScript y los de Python. Los conjuntos de Python comparan los miembros en función de la igualdad, no de la identidad, pero la contrapartida es que los conjuntos de Python sólo permiten miembros inmutables, como las tuplas, y no permiten añadir listas y dicts a los conjuntos.

En la práctica, lo más importante que hacemos con los conjuntos no es añadir y eliminar elementos de ellos, sino comprobar si un valor especificado es un miembro del conjunto. Esto lo hacemos con el método `has()`:

```
let oneDigitPrimes = new Set([2,3,5,7]); oneDigitPrimes. has(2) // => true: 2 es un  
número primo de un dígito oneDigitPrimes. has(3) // => true: también lo es 3  
oneDigitPrimes. has(4) // => false: 4 no es primo oneDigitPrimes. has("5") // => false:  
"5" ni siquiera es un número
```

Lo más importante que hay que entender sobre los conjuntos es que están optimizados para la prueba de pertenencia, y no importa cuántos miembros tenga el conjunto, el método `has()` será muy rápido. El método `includes()` de un array también realiza la prueba de pertenencia, pero el tiempo que tarda es proporcional al tamaño del array, y usar un array como un set puede ser mucho, mucho más lento que usar un objeto Set real.

La clase Set es iterable, lo que significa que puedes utilizar un bucle `for/of` para enumerar todos los elementos de un conjunto:

```
let sum = 0; for(let p of oneDigitPrimes) { // Recorrer los primos de un dígito sum += p;  
// y sumarlos  
} suma // => 17: 2 + 3 + 5 + 7
```

Como los objetos Set son iterables, puedes convertirlos en arrays y listas de argumentos con el operador de extensión ...:

```
[... oneDigitPrimes] // => [2,3,5,7]: el conjunto convertido en un Array Math. max(...  
oneDigitPrimes) // => 7: elementos del conjunto pasados como argumentos de la  
función
```

Los conjuntos se describen a menudo como "colecciones desordenadas". Sin embargo, esto no es exactamente cierto para

la clase Set de JavaScript. Un conjunto de JavaScript no está indexado: no se puede pedir el primer o tercer elemento de un conjunto como se puede hacer con un array. Pero la clase Set de JavaScript siempre recuerda el orden en el que se insertaron los elementos, y siempre utiliza este orden cuando se itera un conjunto: el primer elemento insertado será el primero que se itere (suponiendo que no se haya borrado antes), y el elemento insertado más recientemente será el último que se itere.<sup>2</sup>

Además de ser iterable, la clase Set también implementa un método `forEach()` que es similar al método `array` del mismo nombre:

```
let product = 1; oneDigitPrimes.forEach(n => { product *= n; });
product // => 210: 2 * 3 * 5 * 7
```

El método `forEach()` de un array pasa los índices del array como segundo argumento a la función que se especifique. Los conjuntos no tienen índices, por lo que la versión de este método de la clase Set simplemente pasa el valor del elemento como primer y segundo argumento.

### 11.1.2 La clase Mapa

Un objeto Map representa un conjunto de valores conocidos como *claves*, donde cada clave tiene otro valor asociado (o "mapeado"). En cierto sentido, un mapa es como un array, pero en lugar de utilizar un conjunto de enteros secuenciales como claves, los mapas nos permiten utilizar valores arbitrarios como "índices". Al igual que las matrices, los mapas son rápidos: la búsqueda del valor asociado a una clave será rápida (aunque no tanto como la indexación de una matriz) sin importar el tamaño del mapa.

Crea un nuevo mapa con el constructor Map():

```
let m = new Map(); // Crear un nuevo mapa vacío let n = new Map([ // Un nuevo mapa inicializado con claves de cadena asignadas a números ["uno", 1], ["dos", 2] ]);
```

El argumento opcional del constructor de Map() debe ser un objeto iterable que produzca matrices de dos elementos [clave, valor]. En la práctica, esto significa que si quiere inicializar un mapa cuando lo crea, normalmente escribirá las claves deseadas y los valores asociados como un array de arrays. Pero también puedes utilizar el constructor Map() para copiar otros mapas o para copiar los nombres y valores de las propiedades de un objeto existente:

```
let copy = new Map(n); // Un nuevo mapa con las mismas claves y valores que el mapa n let o = { x: 1, y: 2}; // Un objeto con dos propiedades let p = new Map(Object.entries(o)); // Igual que new map([[ "x", 1], [ "y", 2]])
```

Una vez creado un objeto Map, puede consultar el valor asociado a una clave dada con get() y puede añadir un nuevo par clave/valor con set(). Recuerde, sin embargo, que un mapa es un conjunto de claves, cada una de las cuales tiene un valor asociado. No es lo mismo que un conjunto de pares clave/valor. Si llama a set() con una clave que ya existe en el mapa, cambiará el valor asociado a esa clave, no añadirá un nuevo mapeo clave/valor. Además de get() y set(), la clase Map también define métodos que son como los métodos Set: utilizar has() para comprobar si un mapa incluye la clave especificada; utilizar delete() para eliminar una clave (y su valor asociado) del mapa; utilizar clear() para eliminar todos los pares clave/valor del mapa; y utilizar la propiedad size para averiguar cuántas claves contiene un mapa.

```
let m = new Map(); // Empezar con un mapa vacío  
m.size // => 0: los mapas vacíos no tienen claves  
m.set("uno", 1); // Asignar la clave "uno" al valor 1  
m.set("dos", 2); // Y la clave "dos" al valor 2.  
m.size // => 2: el mapa tiene ahora dos claves  
m.get("dos") // => 2: devuelve el valor asociado a la clave "dos"  
m.get("tres") // => undefined: esta clave no está en el conjunto  
m.set("one", true); // Cambiar el valor asociado a una clave existente  
m.tamaño // => 2: el tamaño no cambia  
m.has("one") // => true: el mapa tiene una clave "one"  
m.has(true) // => false: el mapa no tiene una clave true  
m.delete("one") // => true: la clave existía y el borrado fue exitoso  
m.tamaño // => 1  
m.delete("three") // => false: falló al borrar una clave inexistente  
m.clear(); // Eliminar todas las claves y valores del mapa
```

Al igual que el método add() de Set, el método set() de Map puede encadenarse, lo que permite inicializar los mapas sin necesidad de utilizar arrays de arrays:

```
let m = new Map().set("uno", 1).set("dos", 2).set("tres", 3);  
m.tamaño // => 3  
m.get("dos") // => 2
```

Al igual que con Set, cualquier valor de JavaScript puede utilizarse como clave o valor en un Mapa. Esto incluye nulos, indefinidos y NaN, así como tipos de referencia como objetos y arrays. Y al igual que con la clase Set, Map compara las claves por identidad, no por igualdad, por lo que si se utiliza un objeto o array como clave, se considerará diferente de cualquier otro objeto y array, incluso de aquellos que tengan exactamente las mismas propiedades o elementos:

```
let m = new Map(); // Empieza con un mapa vacío.  
m.set({}, 1); // Asigna un objeto vacío al número 1.  
m.set({}, 2); // Asigna un objeto vacío diferente al número 2.  
m.size // => 2: hay dos claves en este mapa
```

```
m.get({}) //=> undefined: pero este objeto vacío no es una llave  
m.set(m, undefined); // Asignar al mapa mismo el valor undefined.  
m.has(m) //=> true: m es una llave en sí misma  
m.get(m) //=> undefined: el mismo valor que obtendríamos si m no fuera una clave
```

Los objetos Map son iterables, y cada valor iterado es un array de dos elementos donde el primer elemento es una clave y el segundo elemento es el valor asociado a esa clave. Si utilizas el operador spread con un objeto Map, obtendrás un array de arrays como los que pasamos al constructor de Map(). Y al iterar un mapa con un bucle for/of, es idiomático utilizar la asignación de desestructuración para asignar la clave y el valor a variables separadas:

```
let m = new Map([["x", 1], ["y", 2]]); [... m] //=> [["x", 1], ["y",  
2]]  
  
for(let [key, value] of m) {  
    // En la primera iteración, la clave será "x" y el valor será 1  
    // En la segunda iteración, la clave será "y" y el valor será 2  
}
```

Al igual que la clase Set, la clase Map itera en orden de inserción. El primer par clave/valor iterado será el que se haya añadido menos recientemente al mapa, y el último par iterado será el que se haya añadido más recientemente.

Si quiere iterar sólo las claves o sólo los valores asociados de un mapa, utilice los métodos keys() y values(): éstos devuelven objetos iterables que iteran claves y valores, en orden de inserción. (El método entries() devuelve un objeto iterable que itera los pares clave/valor, pero esto es exactamente lo mismo que iterar el mapa directamente).

```
[... m.keys()] //=> ["x", "y"]: sólo las claves
```

```
[... m. valores()] //=> [1, 2]: sólo los valores  
[... m. entries()] //=> [{"x", 1}, {"y", 2}]: igual que [...m]
```

Los objetos del mapa también pueden ser iterados utilizando el método `forEach()` que fue implementado por primera vez por la clase `Array`.

```
m. forEach((valor, clave) => { // anotar valor, clave NO clave, valor  
    // En la primera invocación, el valor será 1 y la clave será "x"  
    // En la segunda invocación, el valor será 2 y la clave será "y" });
```

Puede parecer extraño que el parámetro `valor` vaya antes que el parámetro `clave` en el código anterior, ya que con la iteración `for/of`, la clave va primero. Como se indicó al principio de esta sección, se puede pensar en un mapa como un array generalizado en el que los índices enteros del array se sustituyen por valores clave arbitrarios. El método `forEach()` de los arrays pasa el elemento del array primero y el índice del array después, así que, por analogía, el método `forEach()` de un mapa pasa el valor del mapa primero y la clave del mapa después.

### 11.1.3 WeakMap y WeakSet

La clase `WeakMap` es una variante (pero no una subclase real) de la clase `Map` que no impide que sus valores clave sean recolectados por la basura. La recolección de basura es el proceso por el cual el intérprete de JavaScript recupera la memoria de los objetos que ya no son "alcanzables" y no pueden ser utilizados por el programa. Un mapa normal mantiene referencias "fuertes" a sus valores clave, y siguen siendo accesibles a través del mapa, incluso si todas las demás referencias a ellos han desaparecido. El `WeakMap`, por el contrario, mantiene referencias "débiles" a sus

valores clave para que no sean accesibles a través del WeakMap, y su presencia en el mapa no impide que se recupere su memoria.

El constructor WeakMap() es igual que el constructor Map(), pero hay algunas diferencias significativas entre WeakMap y Map:

- Las claves de WeakMap deben ser objetos o matrices; los valores primitivos no están sujetos a la recolección de basura y no pueden ser utilizados como claves.
- WeakMap sólo implementa los métodos get(), set(), has() y delete(). En particular, WeakMap no es iterable y no define keys(), values(), o forEach(). Si WeakMap fuera iterable, entonces sus claves serían alcanzables y no sería débil.
- Del mismo modo, WeakMap no implementa la propiedad size porque el tamaño de un WeakMap podría cambiar en cualquier momento a medida que los objetos son recolectados por la basura.

El uso previsto de WeakMap es permitirle asociar valores con objetos sin causar fugas de memoria. Supongamos, por ejemplo, que estás escribiendo una función que toma un argumento de objeto y necesita realizar algún cálculo que consume tiempo en ese objeto. Por eficiencia, te gustaría almacenar en caché el valor calculado para su posterior reutilización. Si utilizas un objeto Map para implementar la caché, evitarás que alguno de los objetos sea recuperado, pero si utilizas un WeakMap, evitarás este problema. (A menudo se puede conseguir un resultado similar utilizando una propiedad privada Symbol para almacenar en caché el valor calculado directamente en el objeto. Véase [§6.10.3.](#))

WeakSet implementa un conjunto de objetos que no impide que esos objetos sean recolectados por la basura. El constructor WeakSet() funciona como el constructor Set(), pero los objetos WeakSet difieren de los objetos Set de la misma manera que los objetos WeakMap difieren de los objetos Map:

- WeakSet no permite valores primitivos como miembros.
- WeakSet sólo implementa los métodos add(), has() y delete() y no es iterable.
- WeakSet no tiene una propiedad de tamaño.

WeakSet no se utiliza con frecuencia: sus casos de uso son como los de WeakMap. Si quieras marcar (o "marcar") un objeto como si tuviera alguna propiedad o tipo especial, por ejemplo, puedes añadirlo a un WeakSet. Entonces, cuando quieras comprobar esa propiedad o tipo, puedes comprobar la pertenencia a ese WeakSet. Hacer esto con un conjunto normal impediría que todos los objetos marcados fueran recolectados por la basura, pero esto no es una preocupación cuando se usa WeakSet.

## 11.2 Matrices tipificadas y datos binarios

Las matrices normales de JavaScript pueden tener elementos de cualquier tipo y pueden crecer o reducirse dinámicamente. Las implementaciones de JavaScript realizan muchas optimizaciones para que los usos típicos de las matrices de JavaScript sean muy rápidos. Sin embargo, siguen siendo bastante diferentes de los tipos de arrays de lenguajes de bajo nivel como C y Java. *Las matrices tipadas*, que son nuevas en ES6,<sup>3</sup> se acercan mucho más a las matrices de bajo nivel de esos lenguajes. Los arrays tipificados no son técnicamente arrays (Array.isArray() devuelve false para

ellos), pero implementan todos los métodos de arrays descritos en §7.8 más algunos propios. Sin embargo, difieren de los arrays normales en algunos aspectos muy importantes:

- Los elementos de una matriz tipada son todos números. Sin embargo, a diferencia de los números normales de JavaScript, las matrices tipadas permiten especificar el tipo (enteros con y sin signo y coma flotante IEEE-754) y el tamaño (de 8 a 64 bits) de los números que se almacenan en la matriz.
- Debes especificar la longitud de un array tipado cuando lo creas, y esa longitud nunca puede cambiar.
- Los elementos de un array tipado se inicializan siempre a 0 cuando se crea el array.

### 11.2.1 Tipos de matrices tipificadas

JavaScript no define una clase `TypedArray`. En su lugar, hay 11 clases de arrays tipados, cada una con un tipo de elemento y un constructor diferentes:

| Constructor                      | Tipo numérico                       |
|----------------------------------|-------------------------------------|
| <code>Int8Array()</code>         | bytes con signo                     |
| <code>Uint8Array()</code>        | bytes sin signo                     |
| <code>Uint8ClampedArray()</code> | bytes sin signo sin rollover        |
| <code>Int16Array()</code>        | enteros cortos de 16 bits con signo |
| <code>Uint16Array()</code>       | enteros cortos de 16 bits sin signo |
| <code>Int32Array()</code>        | enteros de 32 bits con signo        |

---

|   |   |
|---|---|
| Uint32Array()   | enteros de 32 bits sin signo                                      |
| BigInt64Array()   | valores BigInt de 64 bits con signo (ES2020)                      |
| BigUint64Array()  | valores BigUint de 64 bits sin signo (ES2020)                     |
| Float32Array()  | Valor de coma flotante de 32 bits                                 |
| Float64Array()  | Valor de coma flotante de 64 bits: un número normal de JavaScript |
| Los tipos cuyos nombres comienzan por Int contienen enteros con signo, de 1, 2 o 4 bytes (8, 16 o 32 bits). Los tipos cuyos nombres comienzan por Uint contienen enteros sin signo de esas mismas longitudes. Los tipos "BigInt" y "BigUint" contienen enteros de 64 bits, representados en JavaScript como valores BigInt (véase <a href="#">§3.2.5</a> ). Los tipos que empiezan por Float contienen números de punto flotante. Los elementos de un Float64Array son del mismo tipo que los números normales de JavaScript. Los elementos de un Float32Array tienen una precisión menor y un rango más pequeño, pero requieren sólo la mitad de la memoria. (Este tipo se llama float en C y Java). |   |

Uint8ClampedArray es una variante de caso especial de Uint8Array.

Ambos tipos contienen bytes sin signo y pueden representar números entre 0 y 255. Con Uint8Array, si almacena un valor mayor que 255 o menor que cero en un elemento del array, éste "se envuelve" y obtiene otro valor. Así es como funciona la memoria del ordenador a bajo nivel, por lo que es muy rápido.

Uint8ClampedArray realiza una comprobación de tipo adicional para que, si almacena un valor mayor que 255 o menor que 0, se

"sujete" a 255 o 0 y no se envuelva. (Este comportamiento de sujeción es requerido por la API de bajo nivel del elemento HTML < canvas> para manipular los colores de los píxeles).

Cada uno de los constructores de matrices tipificadas tiene una propiedad BYTES\_PER\_ELEMENT con el valor 1, 2, 4 u 8, dependiendo del tipo.

### 11.2.2 Creación de matrices tipificadas

La forma más sencilla de crear un array tipado es llamar al constructor apropiado con un argumento numérico que especifique el número de elementos que quieras en el array:

```
let bytes = new Uint8Array(1024); // 1024 bytes let matrix = new Float64Array(9); //  
Una matriz de 3x3 let point = new Int16Array(3); // Un punto en el espacio 3D let rgba  
= new Uint8ClampedArray(4); // Un valor de píxel RGBA de 4 bytes  
let sudoku = new Int8Array(81); // Un tablero de sudoku de 9x9
```

Cuando se crean arrays tipados de esta manera, se garantiza que todos los elementos del array se inicializan a 0, On, o 0.0. Pero si conoces los valores que quieras en tu array tipado, también puedes especificar esos valores cuando creas el array. Cada uno de los constructores de matrices tipificadas tiene métodos de fábrica estáticos from() y of() que funcionan como Array.from() y Array.of():

```
let white = Uint8ClampedArray. of(255, 255, 255, 0); // RGBA blanco opaco
```

Recordemos que el método de fábrica Array.from() espera un objeto tipo array o iterable como primer argumento. Lo mismo ocurre con las variantes tipificadas de los arrays, excepto que el objeto iterable o similar a un array debe tener también elementos

numéricos. Las cadenas son iterables, por ejemplo, pero no tendría sentido pasarlas al método de fábrica from() de un array tipado.

Si sólo está utilizando la versión de un solo argumento de from(), puede omitir el . from y pasar su objeto iterable o tipo array directamente a la función del constructor, que se comporta exactamente igual. Tenga en cuenta que tanto el constructor como el método de fábrica from() le permiten copiar arrays tipados existentes, aunque posiblemente cambiando el tipo:

```
let ints = Uint32Array.from(white); // Los mismos 4 números, pero como ints
```

Cuando se crea un nuevo array tipado a partir de un array, iterable u objeto similar a un array existente, los valores pueden ser truncados para ajustarse a las restricciones de tipo de su array. No hay advertencias ni errores cuando esto sucede:

```
// Flotantes truncados a ints, ints más largos truncados a 8 bits
Uint8Array.of(1.23, 2.99, 45000) //=> new Uint8Array([1, 2,
200])
```

Finalmente, hay una forma más de crear arrays tipados que implica el tipo ArrayBuffer. Un ArrayBuffer es una referencia opaca a un trozo de memoria. Puedes crear uno con el constructor; sólo tienes que pasar el número de bytes de memoria que quieras asignar:

```
let buffer = new ArrayBuffer(1024*1024);
buffer.byteLength //=> 1024*1024; un megabyte de memoria
```

La clase ArrayBuffer no permite leer ni escribir ninguno de los bytes que ha asignado. Pero puedes crear arrays tipados que utilicen la memoria del buffer y que sí te permitan leer y escribir

esa memoria. Para ello, llama al constructor de arrays tipados con un ArrayBuffer como primer argumento, un desplazamiento de bytes dentro del buffer del array como segundo argumento, y la longitud del array (en elementos, no en bytes) como tercer argumento. El segundo y tercer argumento son opcionales. Si se omiten ambos, el array utilizará toda la memoria del buffer del array. Si sólo omites el argumento de la longitud, tu matriz utilizará toda la memoria disponible entre la posición inicial y el final de la matriz. Una cosa más a tener en cuenta sobre esta forma del constructor de arrays tipados: los arrays deben estar alineados con la memoria, por lo que si se especifica un desplazamiento de bytes, el valor debe ser un múltiplo del tamaño de su tipo. El constructor Int32Array() requiere un múltiplo de cuatro, por ejemplo, y Float64Array() requiere un múltiplo de ocho.

Teniendo en cuenta el ArrayBuffer creado anteriormente, podrías crear arrays tipados como estos:

```
let asbytes = new Uint8Array(buffer); // Visto como bytes
let asints = new Int32Array(buffer); // Visto como ints de 32 bits con signo
let lastK = new Uint8Array(buffer, 1023*1024); // Último kilobyte como bytes
let ints2 = new Int32Array(buffer, 1024, 256); // Segundo kilobyte como 256 enteros
```

Estos cuatro arrays tipados ofrecen cuatro vistas diferentes de la memoria representada por el ArrayBuffer. Es importante entender que todos los arrays tipificados tienen un ArrayBuffer subyacente, incluso si no se especifica explícitamente. Si llamas al constructor de un array tipado sin pasarle un objeto buffer, se creará automáticamente un buffer del tamaño adecuado. Como se describe más adelante, la propiedad buffer de cualquier array tipado se refiere a su objeto ArrayBuffer subyacente. La razón para trabajar directamente con los objetos ArrayBuffer es que a veces

puedes querer tener múltiples vistas de matrices tipificadas de un solo buffer.

### 11.2.3. Uso de matrices tipificadas

Una vez que haya creado una matriz tipificada, puede leer y escribir sus elementos con la notación regular de corchetes, tal como lo haría con cualquier otro objeto tipo matriz:

```
// Devuelve el mayor primo menor que n, utilizando el tamiz de Eratóstenes
function sieve(n) { let a = new Uint8Array(n+1); // a[x] será 1 si x es compuesto
let max = Math.floor(Math.sqrt(n)); // No hagas factores superiores a éste
let p = 2; // 2 es el primer primo
while(p <= max) { // Para primos menores que max
for(let i = 2*p; i <= n; i += p)
// Marca los múltiplos de p como compuestos
a[i] = 1; while(a[++p]) /* empty */; // El siguiente índice no marcado es primo
} while(a[n]) n--; // Haz un bucle hacia atrás para encontrar el último primo
return n; // Y devuélvelo
}
```

La función aquí calcula el mayor número primo más pequeño que el número que se especifica. El código es exactamente el mismo que con un array normal de JavaScript, pero el uso de Uint8Array() en lugar de Array() hace que el código se ejecute más de cuatro veces más rápido y utilice ocho veces menos memoria en mis pruebas.

Las matrices tipificadas no son verdaderas matrices, pero reimplementan la mayoría de los métodos de las matrices, por lo que se pueden utilizar prácticamente igual que las matrices normales:

```
let ints = new Int16Array(10); // 10 enteros cortos
ints.fill(3).map(x=> x*x).join("") // => "999999999"
```

Recuerde que los arrays tipados tienen longitudes fijas, por lo que la propiedad length es de sólo lectura, y los métodos que cambian

la longitud del array (como push(), pop(), unshift(), shift() y splice()) no están implementados para los arrays tipados. Los métodos que alteran el contenido de un array sin cambiar su longitud (como sort(), reverse() y fill()) están implementados. Los métodos como map() y slice() que devuelven nuevas matrices devuelven una matriz tipada del mismo tipo que aquella a la que se llama.

#### 11.2.4 Métodos y propiedades de matrices tipificadas

Además de los métodos estándar de las matrices, las matrices tipadas también implementan algunos métodos propios. El método set() establece múltiples elementos de un array tipado a la vez copiando los elementos de un array normal o tipado en un array tipado:

```
let bytes = new Uint8Array(1024); // Un buffer de 1K let pattern = new  
Uint8Array([0,1,2,3]); // Un array de 4 bytes bytes.set(pattern); // Copiarlos al inicio de  
otro array de bytes bytes.set(pattern, 4); // Copiarlos de nuevo en un offset diferente  
bytes.set([0,1,2,3], 8); // O simplemente copiar los valores directamente desde un  
array normal  
bytes.slice(0, 12) // => new Uint8Array([0,1,2,3,0,1,2,3,0,1,2,3])
```

El método set() toma un array o un array tipado como primer argumento y un desplazamiento de elementos como segundo argumento opcional, que por defecto es 0 si se deja sin especificar. Si está copiando valores de un array tipado a otro, la operación será probablemente muy rápida.

Las matrices tipificadas también tienen un método de submatriz que devuelve una parte de la matriz sobre la que se llama:

```
let ints = new Int16Array([0,1,2,3,4,5,6,7,8,9]); // 10 enteros cortos  
let last3 = ints.subarray(ints.length-3, ints.length); // Los 3 últimos last3[0] // => 7:  
estos son los mismos que ints[7]
```

subarray() toma los mismos argumentos que el método slice() y parece funcionar de la misma manera. Pero hay una diferencia importante. slice() devuelve los elementos especificados en una nueva matriz de tipo independiente que no comparte memoria con la matriz original.

subarray() no copia ninguna memoria; sólo devuelve una nueva vista de los mismos valores subyacentes:

```
ints[9] = -1; // Cambia un valor del array original y...  
last3[2] // => -1: también cambia en la submatriz
```

El hecho de que el método subarray() devuelva una nueva vista de un array existente nos devuelve al tema de los ArrayBuffer. Todo array tipado tiene tres propiedades que se relacionan con el buffer subyacente:

```
last3.buffer // El objeto ArrayBuffer para un array tipado last3.buffer === ints.buffer  
// => true: ambas son vistas del mismo buffer last3.byteOffset // => 14: esta vista  
comienza en el byte 14 del buffer  
last3.byteLength // => 6: esta vista tiene 6 bytes (3 ints de 16 bits) long last3.buffer.  
byteLength // => 20: pero el buffer subyacente tiene 20 bytes
```

La propiedad buffer es el ArrayBuffer del array. byteOffset es la posición inicial de los datos del array dentro del buffer subyacente. Y byteLength es la longitud de los datos del array en bytes. Para cualquier tipo de array, a, este invariante debería ser siempre verdadero:

```
a.length * a.BYTES_PER_ELEMENT === a.byteLength // => true
```

Los ArrayBuffers son simplemente trozos opacos de bytes. Puedes acceder a esos bytes con arrays tipificados, pero un ArrayBuffer no es en sí mismo un array tipificado. Sin embargo, ten cuidado: puedes utilizar la indexación numérica de arrays con ArrayBuffers igual que con cualquier objeto de JavaScript. Hacerlo no te da acceso a los bytes del buffer, pero puede causar errores de confusión:

```
let bytes = new Uint8Array(8); bytes[0] = 1; // Establece el primer byte en 1 bytes.  
buffer[0] //=> undefined: el buffer no tiene índice 0 bytes. buffer[1] = 255; // Intenta establecer incorrectamente un byte en el buffer  
bytes.buffer[1] //=> 255: esto sólo establece una propiedad JS regular bytes[1] //  
=> 0: la línea anterior no estableció el byte
```

Anteriormente vimos que se puede crear un ArrayBuffer con el constructor ArrayBuffer() y luego crear arrays tipados que usen ese buffer. Otro enfoque es crear un array tipado inicial, y luego utilizar el buffer de ese array para crear otras vistas:

```
let bytes = new Uint8Array(1024); // 1024 bytes let ints = new Uint32Array(bytes.  
buffer); // o 256 enteros let floats = new Float64Array(bytes.buffer); // o 128  
dóbles
```

## 11.2.5 DataView y Endianness

Las matrices tipificadas permiten ver la misma secuencia de bytes en trozos de 8, 16, 32 o 64 bits. Esto expone la "endiabilidad": el orden en que los bytes se organizan en palabras más largas. Por razones de eficiencia, las matrices tipificadas utilizan la endiabilidad nativa del hardware subyacente. En los sistemas littleendian, los bytes de un número se ordenan en un ArrayBuffer de menor a mayor importancia. En las plataformas big-endian, los bytes se ordenan de más significativo a menos significativo. Puedes

determinar el endianamiento de la plataforma subyacente con un código como este:

```
// Si el entero 0x00000001 está dispuesto en la memoria como 01 00  
00 00, entonces  
// estamos en una plataforma little-endian. En una plataforma big-endian,  
obtendríamos // los bytes 00 00 00 01 en su lugar.  
let littleEndian = new Int8Array(new Int32Array([1]).buffer)  
  
[0] === 1;
```

Hoy en día, las arquitecturas de CPU más comunes son little-endian. Sin embargo, muchos protocolos de red y algunos formatos de archivos binarios requieren un ordenamiento de bytes big-endian. Si estás usando arrays tipados con datos que provienen de la red o de un archivo, no puedes simplemente asumir que la endianidad de la plataforma coincide con el orden de los bytes de los datos. En general, cuando se trabaja con datos externos, se puede utilizar Int8Array y Uint8Array para ver los datos como un array de bytes individuales, pero no se deben utilizar los otros arrays tipados con tamaños de palabra multibyte. En su lugar, puedes utilizar la clase DataView, que define métodos para leer y escribir valores de un ArrayBuffer con un ordenamiento de bytes explícitamente especificado:

```
// Supongamos que tenemos una matriz tipificada de bytes de datos binarios para
// procesar. Primero,
// creamos un objeto DataView para poder leer y escribir de forma flexible //
// valores de esos bytes let view = new DataView(bytes.buffer, bytes.byteOffset,
// bytes.byteLength);

let int = view.getInt32(0); // Leer el int con signo big-endian del byte 0 int = view.
getInt32(4, false); // El siguiente int también es bigendian int = view.getInt32(8, true);
// El siguiente int es little-endian y sin signo view.setUint32(8, int, false); // Escribirlo de
nuevo en formato bigendian
```

DataView define 10 métodos get para cada una de las 10 clases de arrays tipificados (excluyendo Uint8ClampedArray). Tienen nombres como getInt16(), getUint32(), getBigInt64(), y getFloat64(). El primer argumento es el desplazamiento de bytes dentro del ArrayBuffer en el que comienza el valor. Todos estos métodos getter, excepto getInt8() y getUint8(), aceptan un valor booleano opcional como segundo argumento. Si el segundo argumento se omite o es falso, se utiliza el ordenamiento big-endian de los bytes. Si el segundo argumento es verdadero, se utiliza el ordenamiento little-endian.

DataView también define 10 métodos Set correspondientes que escriben valores en el ArrayBuffer subyacente. El primer argumento es el desplazamiento en el que comienza el valor. El segundo argumento es el valor a escribir.

Cada uno de los métodos, excepto setInt8() y setUint8(), acepta un tercer argumento opcional. Si el argumento se omite o es falso, el valor se escribe en formato big-endian con el byte más significativo

primero. Si el argumento es verdadero, el valor se escribe en formato little-endian con el byte menos significativo primero.

Las matrices tipificadas y la clase DataView le ofrecen todas las herramientas necesarias para procesar datos binarios y le permiten escribir programas de JavaScript que hagan cosas como descomprimir archivos ZIP o extraer metadatos de archivos JPEG.

## 11.3 Comparación de patrones con expresiones regulares

Una *expresión regular* es un objeto que describe un patrón textual.

La dirección

La clase RegExp de JavaScript representa expresiones regulares, y tanto String como RegExp definen métodos que utilizan expresiones regulares para realizar potentes funciones de comparación de patrones y de búsqueda y reemplazo en el texto.

Sin embargo, para utilizar la API RegExp de forma efectiva, también debes aprender a describir patrones de texto utilizando la gramática de expresiones regulares, que es esencialmente un mini lenguaje de programación propio. Afortunadamente, la gramática de expresiones regulares de JavaScript es bastante similar a la gramática utilizada por muchos otros lenguajes de programación, por lo que es posible que ya esté familiarizado con ella. (Y si no lo estás, el esfuerzo que inviertas en aprender las expresiones regulares de JavaScript probablemente te será útil también en otros contextos de programación).

Las subsecciones que siguen describen primero la gramática de las expresiones regulares, y luego, después de explicar cómo escribir

expresiones regulares, explican cómo puedes usarlas con los métodos de las clases String y RegExp.

### 11.3.1 Definición de expresiones regulares

En JavaScript, las expresiones regulares se representan mediante objetos RegExp. Los objetos RegExp pueden crearse con el constructor RegExp(), por supuesto, pero es más frecuente que se creen utilizando una sintaxis literal especial. Así como los literales de cadena se especifican como caracteres entre comillas, los literales de expresiones regulares se especifican como caracteres dentro de un par de caracteres de barra (/). Así, su código JavaScript puede contener líneas como ésta:

```
let patrón = /s$/;
```

Esta línea crea un nuevo objeto RegExp y lo asigna a la variable patrón. Este objeto RegExp en particular coincide con cualquier cadena que termine con la letra "s". Esta expresión regular podría haber sido definida de forma equivalente con el constructor RegExp(), así

```
let pattern = new RegExp("s$");
```

Las especificaciones de patrones de expresiones regulares consisten en una serie de caracteres. La mayoría de los caracteres, incluidos todos los caracteres alfanuméricos, simplemente describen los caracteres que deben coincidir literalmente. Así, la expresión regular /java/ coincide con cualquier cadena que contenga la subcadena "java". Otros caracteres de las expresiones regulares no se comparan literalmente, pero tienen un significado

especial. Por ejemplo, la expresión regular `/s$/` contiene dos caracteres. El primero, "s", coincide literalmente. El segundo, "\$", es un metacarácter especial que coincide con el final de una cadena. Así, esta expresión regular coincide con cualquier cadena que contenga la letra "s" como último carácter.

Como veremos, las expresiones regulares también pueden tener uno o más caracteres de bandera que afectan a su funcionamiento. Las banderas se especifican a continuación del segundo carácter de barra en los literales RegExp, o como un segundo argumento de cadena para el constructor RegExp(). Si quisiéramos coincidir con cadenas que terminan con "s" o "S", por ejemplo, podríamos usar la bandera `i` con nuestra expresión regular para indicar que queremos una coincidencia sin distinción de mayúsculas y minúsculas:

```
let patrón = /s$/i;
```

Las siguientes secciones describen los distintos caracteres y metacaracteres utilizados en las expresiones regulares de JavaScript.

## CARACTERES LITERALES

Todos los caracteres alfabéticos y los dígitos coinciden literalmente en las expresiones regulares. La sintaxis de las expresiones regulares de JavaScript también admite ciertos caracteres no alfabéticos mediante secuencias de escape que comienzan con una barra invertida (`\`). Por ejemplo, la secuencia `\n` coincide con un carácter literal de nueva línea en una cadena. [La Tabla 11-1](#) enumera estos caracteres.

*Tabla 11-1. Caracteres literales de expresiones regulares*

| Carácter                                       | Partidos  |
|--|---|
|  | Su mismo  |
| Carácter alfanumérico                          |   |
| \0   | El carácter NUL (\u0000)  |
| \t   | Ficha (\u0009)  |
| \n   | Nueva línea (\u000A)  |
| \v   | Pestaña vertical (\u000B)   |
| \f   | Alimentación del formulario (\u000C)  |
| \r   | Retorno de carro (\u000D)   |
| \N - La vida de los niños en la escuela        | El carácter latino especificado por el número hexadecimal <i>nn</i> ; por ejemplo, \xA es lo mismo que \n.  |
| \uxxxx   | El carácter Unicode especificado por el número hexadecimal <i>xxxx</i> ; por ejemplo, \u0009 es lo mismo que \t.  |
| \N - La vida en el mundo es un juego de niños. | El carácter Unicode especificado por el punto de código <i>n</i> , donde <i>n</i> es de uno a seis dígitos hexadecimales entre 0 y 10FFFF. Tenga en cuenta que esta sintaxis sólo se admite en las expresiones regulares que utilizan el indicador u. |
| \cX  | El carácter de control ^X; por ejemplo, \cJ equivale al carácter de nueva línea \n.   |

Algunos caracteres de puntuación tienen un significado especial en las expresiones regulares. Son:

```
^ $ . * + ? = ! : | \ / ()[]{} }
```

Los significados de estos caracteres se discuten en las secciones siguientes. Algunos de estos caracteres tienen un significado especial sólo dentro de ciertos contextos de una expresión regular y se tratan literalmente en otros contextos. Sin embargo, como regla general, si desea incluir cualquiera de estos caracteres de puntuación literalmente en una expresión regular, debe precederlos con un \. Otros caracteres de puntuación, como las comillas y la @, no tienen un significado especial y simplemente se combinan literalmente en una expresión regular.

Si no puede recordar exactamente qué caracteres de puntuación deben escaparse con una barra invertida, puede colocar con seguridad una barra invertida antes de cualquier carácter de puntuación. Por otro lado, tenga en cuenta que muchas letras y números tienen un significado especial cuando van precedidos de una barra invertida, por lo que cualquier letra o número que quiera que coincida literalmente no debe escaparse con una barra invertida. Para incluir un carácter de barra invertida literalmente en una expresión regular, debe escapar con una barra invertida, por supuesto. Por ejemplo, la siguiente expresión regular coincide con cualquier cadena que incluya una barra invertida: /\\\N/. (Y si utiliza el constructor RegExp(), tenga en cuenta que cualquier barra invertida en su expresión regular debe ser duplicada, ya que las cadenas también utilizan barras invertidas como carácter de escape).

## CLASES DE CARACTERES

Los caracteres literales individuales pueden combinarse en *clases de caracteres* colocándolos entre corchetes. Una clase de caracteres coincide con cualquier carácter que esté contenido en ella. Así, la expresión regular /[abc]/ coincide con cualquiera de las letras a, b o c. También pueden definirse clases de caracteres negados, que coinciden con cualquier carácter excepto los contenidos dentro de los corchetes. Una clase de caracteres negada se especifica colocando un signo de intercalación (^) como primer carácter dentro del corchete izquierdo.

La dirección

RegExp /^[^abc]/ coincide con cualquier carácter que no sea a, b o c. Las clases de caracteres pueden utilizar un guión para indicar un rango de caracteres. Para coincidir con cualquier carácter en minúscula del alfabeto latino, utilice /[az]/, y para coincidir con cualquier letra o dígito del alfabeto latino, utilice /[a-zA-Z0-9]/. (Y si desea incluir un guión real en su clase de caracteres, simplemente hágalo el último carácter antes del corchete derecho).

Debido a que ciertas clases de caracteres son de uso común, la sintaxis de expresiones regulares de JavaScript incluye caracteres especiales y secuencias de escape para representar estas clases comunes. Por ejemplo, \s coincide con el carácter de espacio, el carácter de tabulación y cualquier otro carácter de espacio en blanco Unicode; \S coincide con cualquier carácter que *no* sea un espacio en blanco Unicode. [La Tabla 11-2](#) enumera estos caracteres y resume la sintaxis de las clases de caracteres. (Tenga en cuenta que varias de estas secuencias de escape de clase de caracteres sólo coinciden con caracteres ASCII y no se han

extendido para trabajar con caracteres Unicode. Sin embargo, puede definir explícitamente sus propias clases de caracteres Unicode; por ejemplo, `/[\u0400-\u04FF]/` coincide con cualquier carácter cirílico).

*Tabla 11-2. Clases de caracteres de expresiones regulares*

|              |   |
|--------------|---|
| <b>Ch</b>    |   |
| <b>ara</b>   |   |
| <b>cte r</b> |   |
|              | <b>Partidos</b>   |
| [.           | Un carácter cualquiera entre los paréntesis.<br>..<br>]   |
| [^ ..        | Cualquier carácter que no esté entre los paréntesis.<br>.]  |
| .            | Cualquier carácter excepto la nueva línea u otro terminador de línea Unicode. O, si la RegExp utiliza el indicador s, entonces un punto coincide con cualquier carácter, incluidos los terminadores de línea. |
| \w           | Cualquier carácter de palabra ASCII. Equivale a [a-zA-Z0-9_].   |
| \W           | Cualquier carácter que no sea un carácter de palabra ASCII. Equivale a [^a-zA-Z0-9_].   |
| \s           | Cualquier carácter Unicode de espacio en blanco.  |
| \S           | Cualquier carácter que no sea un espacio en blanco Unicode.   |
| \d           | Cualquier dígito ASCII. Equivale a [0-9].   |
| \D           | Cualquier carácter que no sea un dígito ASCII. Equivale a [^0-9].   |

Un retroceso literal (caso especial).  
[\b]

Tenga en cuenta que los escapes de clase de caracteres especiales pueden utilizarse entre corchetes. \s coincide con cualquier carácter de espacio en blanco, y \d coincide con cualquier dígito, por lo que /[\s\d]/ coincide con cualquier carácter de espacio en blanco o dígito. Tenga en cuenta que hay un caso especial. Como verás más adelante, el escape \b tiene un significado especial. Sin embargo, cuando se utiliza dentro de una clase de caracteres, representa el carácter de retroceso. Así, para representar un carácter de retroceso literalmente en una expresión regular, utilice la clase de caracteres con un elemento /[\b]/.

## CLASES DE CARACTERES UNICODE

En ES2018, si una expresión regular utiliza `\p{...}` entonces las clases de caracteres `\P{...}` y su negación `\P{...}` son compatibles. (A partir de principios de 2020, esto está implementado por Node, Chrome, Edge y Safari, pero no en Firefox). Estas clases de caracteres se basan en las propiedades definidas por el estándar Unicode, y el conjunto de caracteres que representan puede cambiar a medida que Unicode evoluciona.

El `\d` sólo coincide con dígitos ASCII. Si desea que coincida con un dígito decimal de cualquiera de los sistemas de escritura del mundo, puede utilizar `\p{Número_decimal}/u`. Y si quieres coincidir con cualquier carácter que es *no* un dígito decimal en cualquier idioma, se puede poner `\p{No_Número_decimal}/u`. Si desea que coincida con cualquier carácter de tipo numérico, incluyendo fracciones y números romanos, puede utilizar `\p{Número}`. Tenga en cuenta que "Decimal\_Number" y "Number" no son específico de JavaScript o de la gramática de las expresiones regulares: es el nombre de una categoría de caracteres definido por el estándar Unicode.

El `\w` sólo funciona para texto ASCII, pero con `\p{Alfabético}/u` podemos aproximar una versión internacionalizada como esta:

```
/[\p{Alfabético}\p{NúmeroDecimal}\p{Marca}]/u
```

(Aunque para ser totalmente compatible con la complejidad de las lenguas del mundo, realmente necesitamos añadir las categorías "Conector\_Puntuación" y "Join\_Control" también).

Como último ejemplo, el `\p` también nos permite definir expresiones regulares que coincidan con caracteres de un alfabeto o escritura particular:

```
let greekLetter = /\p{Script=Greek}/u;
let cyrillicLetter = /\p{Script=Cyrillic}/u;
```

## REPETICIÓN

Con la sintaxis de expresiones regulares que ha aprendido hasta ahora, puede describir un número de dos dígitos como `/\d\d/` y un número de cuatro dígitos como `/\d\d\d\d/`. Pero no tiene ninguna forma de describir, por ejemplo, un número que puede tener cualquier número de dígitos o una cadena de tres letras seguida de un dígito opcional. Estos patrones más complejos utilizan una sintaxis de expresión regular que especifica cuántas veces puede repetirse un elemento de una expresión regular.

Los caracteres que especifican la repetición siguen siempre el patrón al que se aplican. Debido a que ciertos tipos de repetición se utilizan con bastante frecuencia, existen caracteres especiales para representar estos casos. Por ejemplo, + coincide con una o más ocurrencias del patrón anterior.

La Tabla 11-3 resume la sintaxis de repetición.

*Tabla 11-3. Caracteres de repetición de expresiones regulares*

| Carácter | Significado  |
|----------|--|
| {n,m}    | Coincidir con el elemento anterior al menos <i>n</i> veces pero no más de <i>m</i> veces.                                |
| {n,}     | Coincidir con el elemento anterior <i>n</i> o más veces.   |
| {n}      | Coincidir exactamente con <i>n</i> ocurrencias del elemento anterior.  |
| ?        | Coincidir con cero o una ocurrencia del elemento anterior. Es decir, el elemento anterior es opcional. Equivale a {0,1}. |
| +        | Coincidir con una o más ocurrencias del elemento anterior. Equivale a {1,}.  |
| *        | Coincidir con cero o más apariciones del elemento anterior. Equivale a {0,}.   |

Las siguientes líneas muestran algunos ejemplos:

```
let r = /\d{2,4}/; // Coincide con entre dos y cuatro dígitos r = /\w{3}\d?/; // Coincide con exactamente tres caracteres de palabra y un dígito opcional r = /\s+java\s+/; // Coincide con "java" con uno o más espacios antes y después  
r = /[^\s]*/; // Coincide con cero o más caracteres que no son paréntesis abiertos
```

Tenga en cuenta que en todos estos ejemplos, los especificadores de repetición se aplican al único carácter o clase de caracteres que los precede. Si quiere hacer coincidir repeticiones de expresiones más complicadas, tendrá que definir un grupo con paréntesis, que se explican en las siguientes secciones.

Tenga cuidado al utilizar los caracteres de repetición \* y ? Dado que estos caracteres pueden coincidir con cero instancias de lo que les precede, no pueden coincidir con nada. Por ejemplo, la expresión regular /a\*/ en realidad coincide con la cadena "bbbb" porque la cadena contiene cero apariciones de la letra a.

## REPETICIÓN SIN AVARICIA

Los caracteres de repetición listados en la Tabla 11-3 coinciden el mayor número de veces posible mientras permiten que las siguientes partes de la expresión regular coincidan. Decimos que esta repetición es "codiciosa". También es posible especificar que la repetición se haga de forma no codiciosa. Basta con seguir el carácter o caracteres de repetición con un signo de interrogación: ??, +?, \*?, o incluso {1,5}?. Por ejemplo, la expresión regular /a+/ coincide con una o más apariciones de la letra a. Cuando se aplica a la cadena "aaa", coincide con las tres letras. Sin embargo, /a+?/ coincide con una o más apariciones de la letra a, con el menor número de caracteres necesario. Cuando se aplica a la misma cadena, este patrón sólo coincide con la primera letra a.

El uso de la repetición sin avaricia no siempre produce los resultados esperados. Considere el patrón /a+b/, que coincide con una o más a, seguidas de la letra b. Cuando se aplica a la cadena "aaab", coincide con toda la cadena. Ahora utilicemos la versión no

codiciosa: `/a+?b/`. Esto debería coincidir con la letra b precedida por el menor número posible de a's. Cuando se aplica a la misma cadena "aaab", se podría esperar que coincidiera sólo con una a y la última letra b. De hecho, sin embargo, este patrón coincide con toda la cadena, al igual que la versión codiciosa del patrón. Esto se debe a que la coincidencia de patrones de expresiones regulares se realiza encontrando la primera posición de la cadena en la que es posible una coincidencia. Dado que una coincidencia es posible a partir del primer carácter de la cadena, las coincidencias más cortas que comienzan en los caracteres siguientes ni siquiera se consideran.

## ALTERNANCIA, AGRUPACIÓN Y REFERENCIAS

La gramática de las expresiones regulares incluye caracteres especiales para especificar alternativas, agrupar subexpresiones y hacer referencia a subexpresiones anteriores. El carácter | separa las alternativas. Por ejemplo, `/ab|cd|ef/` coincide con la cadena "ab" o la cadena "cd" o la cadena "ef". Y `/\d{3}|[a-z]{4}/` coincide con tres dígitos o cuatro letras minúsculas.

Tenga en cuenta que las alternativas se consideran de izquierda a derecha hasta que se encuentra una coincidencia. Si la alternativa de la izquierda coincide, la alternativa de la derecha se ignora, incluso si hubiera producido una coincidencia "mejor". Así, cuando el patrón `/a|ab/` se aplica a la cadena "ab", sólo coincide con la primera letra.

Los paréntesis tienen varios propósitos en las expresiones regulares. Uno de ellos es agrupar elementos separados en una sola subexpresión para que los elementos puedan ser tratados

como una sola unidad mediante |, \*, +, ?, etc. Por ejemplo, /java(script)?/ coincide con "java" seguido del "script" opcional. Y /(ab|cd)+|ef/ coincide con la cadena "ef" o con una o más repeticiones de cualquiera de las cadenas "ab" o "cd".

Otro propósito de los paréntesis en las expresiones regulares es definir subpatrones dentro del patrón completo. Cuando una expresión regular coincide con éxito con una cadena de destino, es posible extraer las partes de la cadena de destino que coinciden con cualquier sub-patrón entre paréntesis en particular. (Verá cómo se obtienen estas subcadenas coincidentes más adelante en esta sección). Por ejemplo, suponga que está buscando una o más letras minúsculas seguidas de uno o más dígitos. Podría utilizar el patrón /[a-z]+\d+/. Pero suponga que sólo le interesan los dígitos al final de cada coincidencia. Si pone esa parte del patrón entre paréntesis (/[a-z]+(\d+)/), puede extraer los dígitos de cualquier coincidencia que encuentre, como se explica más adelante.

Un uso relacionado de las subexpresiones con paréntesis es permitirle referirse a una subexpresión más adelante en la misma expresión regular. Esto se hace siguiendo a un carácter de paréntesis por un dígito o dígitos. Los dígitos se refieren a la posición de la subexpresión entre paréntesis dentro de la expresión regular. Por ejemplo, \1 se refiere a la primera subexpresión, y \3 a la tercera. Tenga en cuenta que, como las subexpresiones pueden estar anidadas dentro de otras, lo que se cuenta es la posición del paréntesis izquierdo. En la siguiente expresión regular, por ejemplo, la subexpresión anidada ([Ss]cript) se denomina \2:

```
/([Jj]ava([Ss]cript)?)\N-(fun\w*)/
```

Una referencia a una subexpresión anterior de una expresión regular *no* se refiere al patrón de esa subexpresión, sino al texto que coincide con el patrón. Por lo tanto, las referencias pueden utilizarse para imponer la restricción de que porciones separadas de una cadena contengan exactamente los mismos caracteres. Por ejemplo, la siguiente expresión regular coincide con cero o más caracteres entre comillas simples o dobles. Sin embargo, no requiere que las comillas de apertura y cierre coincidan (es decir, ambas comillas simples o ambas comillas dobles):

```
/[""][^"]*[""]/
```

Para exigir que las comillas coincidan, utilice una referencia:

```
/([""])[^"]*\1/
```

El \1 coincide con lo que la primera subexpresión entre paréntesis coincide. En este ejemplo, aplica la restricción de que la comilla de cierre coincide con la comilla de apertura. Esta expresión regular no permite comillas simples dentro de cadenas con comillas dobles o viceversa. (No es legal utilizar una referencia dentro de una clase de caracteres, por lo que no se puede escribir:

```
/([""])[^\\1]*\\1/.)
```

Cuando cubramos la API RegExp más adelante, verás que este tipo de referencia a una subexpresión entre paréntesis es una poderosa característica de las operaciones de búsqueda y reemplazo de expresiones regulares.

También es posible agrupar elementos en una expresión regular sin crear una referencia numerada a esos elementos. En lugar de

agrupar simplemente los elementos dentro de ( y ), comience el grupo con ?: y termine con ). Considere el siguiente patrón:

```
/([Jj]ava(?:[Ss]cript)?)\Nsis(fun\w*)/
```

En este ejemplo, la subexpresión (?:[Ss]cript) se utiliza simplemente para agrupar, por lo que el carácter de repetición ? puede aplicarse al grupo. Estos paréntesis modificados no producen una referencia, por lo que en esta expresión regular, \2 se refiere al texto coincidente con (fun\w\*).

La Tabla 11-4 resume los operadores de alternancia, agrupación y referenciación de expresiones regulares.

*Tabla 11-4. Alternancia de expresiones regulares, agrupación y caracteres de referencia*

| C h         |  |
|-------------|--|
| a           |  |
| r           |  |
| a c         |  |
| t           |  |
| e           |  |
| r           |  |
| Significado |  |
|             | Alternancia: coincide con la subexpresión de la izquierda o con la de la derecha.  |
| ( . )       | Agrupación: agrupa los elementos en una sola unidad que puede utilizarse con *, +, ?,  , etc. También recuerda los caracteres que coinciden con este grupo para usarlos con referencias posteriores. |
| :           |  |
| .           |  |
|             |  |
| (           | Sólo agrupación: agrupa los elementos en una sola unidad, pero no recuerda los caracteres... que coinciden con este grupo.   |
| :           |  |
| .           |  |
|             |  |

## GRUPOS DE CAPTURA CON NOMBRE

ES2018 estandariza una nueva característica que puede hacer que las expresiones regulares sean más auto-documentadas y más fáciles de entender. Esta nueva característica se conoce como "grupos de captura con nombre" y nos permite asociar un nombre a cada paréntesis izquierdo de una expresión regular para poder referirnos al texto coincidente por su nombre en lugar de por su número. Igualmente importante: el uso de nombres permite a alguien que lea el código entender más fácilmente el propósito de esa parte de la expresión regular. A principios de 2020, esta función está implementada en Node, Chrome, Edge y Safari, pero todavía no en Firefox.

Para nombrar un grupo, utilice (?<...>) en lugar de ( y ponga el nombre entre los corchetes. Por ejemplo, ésta es una expresión regular que podría utilizarse para comprobar el formato de la última línea de una dirección postal de Estados Unidos:

```
/(?<ciudad>\w+) (?<estado>[A-Z]{2}) (?<código postal>\d{5})(?<código postal9>-\d{4})?/
```

Fíjese en el contexto que proporcionan los nombres de los grupos para facilitar la comprensión de la expresión regular. En §11.3.2, cuando hablamos de los métodos String replace() y match() y del método

RegEx exec(), verá cómo la API RegEx le permite referirse al texto que coincide con cada uno de estos grupos por su nombre en lugar de por su posición.

Si quiere referirse a un grupo de captura con nombre dentro de una expresión regular, también puede hacerlo por nombre. En el ejemplo anterior, pudimos usar una expresión regular "backreference" para escribir una RegEx que coincidiera con una cadena de comillas simples o dobles donde las comillas abiertas y cerradas tuvieran que coincidir. Podríamos reescribir esta RegEx usando un grupo de captura con nombre y una retro-referencia con nombre como esta:

```
/(?<cita>["])[^\"]*\k<cita>/
```

La \k<cita> es una retro-referencia con nombre al grupo con nombre que captura la comilla abierta.

)

- \n     Coincidirán con los mismos caracteres que se compararon cuando el grupo número *n* se comparó por primera vez. Los grupos son subexpresiones dentro de paréntesis (posiblemente anidados). Los números de grupo se asignan contando los paréntesis de izquierda a derecha. Los grupos formados con ?: no están numerados.

## ESPECIFICAR LA POSICIÓN DEL PARTIDO

Como se ha descrito anteriormente, muchos elementos de una expresión regular coinciden con un solo carácter de una cadena. Por ejemplo, \s coincide con un solo carácter de espacio en blanco. Otros elementos de expresiones regulares coinciden con las posiciones entre los caracteres en lugar de los caracteres reales. \b, por ejemplo, coincide con un límite de palabra ASCII-el límite entre un \w (carácter de palabra ASCII) y un \W (carácter de no-palabra), o el límite entre un carácter de palabra ASCII y el principio o el final de una cadena.<sup>4</sup> Los elementos como \b no

especifican ningún carácter que se utilice en una cadena coincidente; lo que sí especifican, sin embargo, son las posiciones legales en las que puede producirse una coincidencia. A veces estos elementos se denominan *anclas de expresiones regulares* porque anclan el patrón a una posición específica en la cadena de búsqueda. Los elementos de anclaje más utilizados son ^, que ata el patrón al principio de la cadena, y \$, que ancla el patrón al final de la cadena.

Por ejemplo, para buscar la palabra "JavaScript" en una línea por sí misma, puede utilizar la expresión regular /`^JavaScript$`/ . Si quiere buscar "Java" como una palabra por sí misma (no como un prefijo, como está en "JavaScript"), puede probar el patrón /`\sJava\s`/, que requiere un espacio antes y después de la palabra. Pero hay dos problemas con esta solución. En primer lugar, no coincide con "Java" al principio o al final de una cadena, sino sólo si aparece con un espacio a cada lado. En segundo lugar, cuando este patrón encuentra una coincidencia, la cadena coincidente que devuelve tiene espacios iniciales y finales, que no es exactamente lo que se necesita. Así que en lugar de coincidir con los caracteres de espacio reales con \s, coincide (o se ancla a) los límites de las palabras con \b. La expresión resultante es /`\bJava\b`/ . El elemento \B ancla la coincidencia a un lugar que no es un límite de palabra. Así, el patrón /`\B[Ss]cript`/ coincide con "JavaScript" y "postscript", pero no con "script" o "Scripting".

También puede utilizar expresiones regulares arbitrarias como condiciones de anclaje. Si incluye una expresión dentro de caracteres (?= y ), se trata de una condición de anclaje, y especifica que los caracteres encerrados deben coincidir, sin llegar a hacerlo.

Por ejemplo, para que coincida con el nombre de un lenguaje de programación común, pero sólo si va seguido de dos puntos, puede utilizar /[Jj]ava([Ss]cript)?(?=\:)/. Este patrón coincide con la palabra "JavaScript" en "JavaScript: The Definitive Guide", pero no coincide con "Java" en "Java in a Nutshell" porque no va seguida de dos puntos.

Si, en cambio, introduce una aserción con (?!), se trata de una aserción de búsqueda negativa, que especifica que los siguientes caracteres no deben coincidir. Por ejemplo, /Java(?!\Script)([A-Z]\w\*)/ coincide con "Java" seguido de una letra mayúscula y cualquier número de caracteres de palabra ASCII adicionales, siempre que "Java" no vaya seguido de "Script". Coincide con "JavaBeans" pero no con "javanés", y coincide con "JavaScrip" pero no con "JavaScript" o "JavaScripter".

La Tabla 11-5 resume las anclas de las expresiones regulares.

*Tabla 11-5. Caracteres de anclaje de expresiones regulares*

| C h<br>ar a<br>ct er | Significado  |
|----------------------|--|
| ^                    | Consegue el principio de la cadena o, con la bandera m, el principio de una línea.   |
| \$                   | Coincide con el final de la cadena y, con la bandera m, con el final de una línea.   |
| Coincidir            | con el límite de una palabra. Es decir, coincide con la posición entre un carácter \w y un carácter ba \W o entre un carácter \w y el principio o el final de una cadena.<br>(Tenga en cuenta, sin embargo, que [\b] coincide con el retroceso). |
| \B                   | Coincidir con una posición que no sea un límite de palabra.  |

( Una afirmación positiva de lookahead. Requiere que los siguientes caracteres coincidan con el patrón `? p`, pero no incluya esos caracteres en la coincidencia.

=p  
)

Una aserción negativa de lookahead. Requiere que los siguientes caracteres no coincidan con el patrón `p`.

?! p  
)

## AFIRMACIONES DE LOOKBEHIND

ES2018 amplía la sintaxis de las expresiones regulares para permitir afirmaciones "lookbehind". Éstas son como el lookahead pero se refieren al texto anterior a la posición actual del partido. A partir de principios de 2020, se implementan en Node, Chrome y Edge, pero no en Firefox o Safari.

Especifique una afirmación positiva de lookbehind `<?= [A-Z]{2}` y una afirmación negativa de lookbehind `<!....`. Por ejemplo, si se trabaja con direcciones postales de EE.UU., se puede hacer coincidir un código postal de 5 dígitos pero sólo cuando sigue a una abreviatura de dos letras del estado, así:

`/(?<= [A-Z]{2} )\Nd{5}/`

Y se puede hacer coincidir una cadena de dígitos que no esté precedida por un símbolo de moneda Unicode con un A la afirmación de la mirada negativa le gusta esto:

`/(?<![\p{symbolo de la moneda}\d.])\d+(\d+)?/u`

## BANDERAS

Cada expresión regular puede tener uno o más indicadores asociados para alterar su comportamiento de coincidencia.

JavaScript define seis banderas posibles, cada una de las cuales está representada por una sola letra. Los indicadores se especifican después del segundo carácter / de un literal de expresión regular o como una cadena que se pasa como segundo argumento al constructor `RegExp()`. Los indicadores admitidos y sus significados son los siguientes

*g*

La bandera *g* indica que la expresión regular es "global", es decir, que pretendemos utilizarla para encontrar todas las coincidencias dentro de una cadena en lugar de encontrar sólo la primera coincidencia. Esta bandera no altera la forma en que se realiza la coincidencia de patrones, pero, como veremos más adelante, sí altera el comportamiento del método `String match()` y del método `RegExp exec()` de forma importante.

*i*

La bandera *i* especifica que la coincidencia de patrones debe distinguir entre mayúsculas y minúsculas.

*m*

La bandera *m* especifica que la coincidencia debe hacerse en modo "multilínea". Dice que la `RegExp` se utilizará con cadenas multilínea y que las anclas `^` y `$` deben coincidir tanto con el principio y el final de la cadena como con el principio y el final de las líneas individuales dentro de la cadena.

*s*

Al igual que la bandera *m*, la bandera *s* también es útil cuando se trabaja con texto que incluye nuevas líneas. Normalmente, un `".."` en una expresión regular coincide con cualquier carácter, excepto con un terminador de línea. Sin embargo, cuando se utiliza la bandera *s*, `".."` coincide con cualquier carácter, incluidos los terminadores de línea. La bandera *s* se añadió a JavaScript en ES2018 y, desde principios de 2020, es compatible con Node, Chrome, Edge y Safari, pero no con Firefox.

*u*

La bandera *u* significa Unicode, y hace que la expresión regular coincida con puntos de código Unicode completos en lugar de coincidir con valores de 16 bits. Esta bandera fue introducida en

ES6, y debería acostumbrarse a usarla en todas las expresiones regulares a menos que tenga alguna razón para no hacerlo. Si no usa esta bandera, sus RegExps no funcionarán bien con texto que incluya emoji y otros caracteres (incluyendo muchos caracteres chinos) que requieren más de 16 bits. Sin la bandera u, el carácter "." coincide con cualquier valor de 1 UTF-16 de 16 bits. Sin embargo, con la bandera, "." coincide con un punto de código Unicode, incluidos los que tienen más de 16 bits. Establecer la bandera u en un RegExp también le permite utilizar la nueva secuencia de escape \u{...} para el carácter Unicode y también permite la notación \p{...} para las clases de caracteres Unicode.

y

La bandera y indica que la expresión regular es "pegajosa" y debe coincidir con el principio de una cadena o con el primer carácter que sigue a la coincidencia anterior. Cuando se utiliza con una expresión regular que está diseñada para encontrar una única coincidencia, trata efectivamente esa expresión regular como si comenzara con ^ para anclarla al principio de la cadena. Esta bandera es más útil con expresiones regulares que se utilizan repetidamente para encontrar todas las coincidencias dentro de una cadena. En este caso, provoca un comportamiento especial del método String match() y del método RegExp exec() para forzar que cada coincidencia posterior se ancle a la posición de la cadena en la que terminó la última.

Estos indicadores pueden especificarse en cualquier combinación y en cualquier orden. Por ejemplo, si quiere que su expresión regular sea compatible con Unicode para hacer coincidencias sin distinción de mayúsculas y minúsculas y pretende utilizarla para encontrar múltiples coincidencias dentro de una cadena, especificaría las banderas uig, gui o cualquier otra permutación de estas tres letras.

## 11.3.2 Métodos de cadenas para la comparación de patrones

Hasta ahora, hemos estado describiendo la gramática utilizada para definir expresiones regulares, pero no hemos explicado cómo esas expresiones regulares pueden ser utilizadas realmente en el código JavaScript. Ahora pasamos a cubrir la API para utilizar los objetos RegExp. Esta sección comienza explicando los métodos de cadena que utilizan expresiones regulares para realizar operaciones de coincidencia de patrones y de búsqueda y reemplazo. Las secciones que siguen a esta continúan la discusión de la concordancia de patrones con expresiones regulares de JavaScript discutiendo el objeto RegExp y sus métodos y propiedades.

### BUSCAR()

Las cadenas soportan cuatro métodos que utilizan expresiones regulares. El más sencillo es search(). Este método toma un argumento de expresión regular y devuelve la posición del carácter del inicio de la primera subcadena coincidente o -1 si no hay coincidencia:

```
"JavaScript".search(/script/ui) // => 4  
"Python".search(/script/ui) // => -1
```

Si el argumento de search() no es una expresión regular, primero se convierte en una pasándola al constructor RegExp. search() no admite búsquedas globales; ignora la bandera g de su argumento de expresión regular.

## REPLACE()

El método replace() realiza una operación de búsqueda y reemplazo. Toma una expresión regular como primer argumento y una cadena de reemplazo como segundo argumento. Busca en la cadena a la que se llama las coincidencias con el patrón especificado. Si la expresión regular tiene activada la bandera g, el método replace() sustituye todas las coincidencias de la cadena por la cadena de sustitución; en caso contrario, sólo sustituye la primera coincidencia que encuentra. Si el primer argumento de replace() es una cadena en lugar de una expresión regular, el método busca esa cadena literalmente en lugar de convertirla en una expresión regular con el constructor RegExp(), como hace search(). Como ejemplo, puede utilizar replace() de la siguiente manera para proporcionar una capitalización uniforme de la palabra "JavaScript" a lo largo de una cadena de texto:

```
// No importa cómo se escriba en mayúsculas, sustítuyalo por la mayúscula correcta
text.replace(/javascript/gi, "JavaScript");
```

Sin embargo, replace() es más potente que esto. Recuerde que las subexpresiones entre paréntesis de una expresión regular se numeran de izquierda a derecha y que la expresión regular recuerda el texto con el que coincide cada subexpresión. Si aparece un \$ seguido de un dígito en la cadena de sustitución, replace() sustituye esos dos caracteres por el texto que coincide con la subexpresión especificada. Esta función es muy útil. Puede utilizarla, por ejemplo, para sustituir las comillas de una cadena por otros caracteres:

```
// Una cita es una comilla, seguida de cualquier número de
// caracteres de marca de no comillas (que capturamos), seguidos de
// por otra comilla.
let quote = /"(^[^"]*)"/g;
// Reemplazar las comillas rectas por guillemets // dejando el texto citado
// (almacenado en $1) sin cambios.
'Dijo "para"'.replace(quote, "$1") // => 'Dijo
"stop"
```

Si su RegExp utiliza grupos de captura con nombre, entonces puede referirse al texto coincidente por nombre en lugar de por número:

```
let quote = /"(?<quotedText>[^"]*)"/g;
'Ha dicho "para"'.replace(quote, "$<quotedText>") // => 'Ha dicho "para"'
```

En lugar de pasar una cadena de reemplazo como segundo argumento a replace(), también puede pasar una función que será invocada para calcular el valor de reemplazo. La función de reemplazo se invoca con varios argumentos. El primero es el texto completo que coincide. Luego, si la RegExp tiene grupos de captura, entonces las subcadenas que fueron capturadas por esos grupos se pasan como argumentos. El siguiente argumento es la posición dentro de la cadena en la que se encontró la coincidencia. Después, se pasa la cadena completa sobre la que se llamó a replace(). Y finalmente, si el RegExp contenía algún grupo de captura con nombre, el último argumento de la función de reemplazo es un objeto cuyos nombres de propiedades coinciden con los nombres de los grupos de captura y cuyos valores son el texto coincidente. Como ejemplo, aquí está el código que utiliza una función de reemplazo para convertir enteros decimales en una cadena a hexadecimal:

```
dejemos s = "15 por 15 es 225";
s.replace(/\d+/gu, n => parseInt(n).toString(16)) // => "f por f es e1"
```

## MATCH()

El método `match()` es el más general de los métodos de expresión regular de cadenas. Toma una expresión regular como único argumento (o convierte su argumento en una expresión regular pasándola al método

`RegExp()`) y devuelve una matriz que contiene los resultados

de la coincidencia, o `null` si no se encuentra ninguna coincidencia. Si la expresión regular tiene activada la bandera `g`, el método devuelve una matriz con todas las coincidencias que aparecen en la cadena. Por ejemplo:

```
"7 más 8 es igual a 15". match(\d+/g) //=> ["7", "8", "15"]
```

Si la expresión regular no tiene activada la bandera `g`, `match()` no realiza una búsqueda global; simplemente busca la primera coincidencia. En este caso no global, `match()` sigue devolviendo una matriz, pero los elementos de la matriz son completamente diferentes. Sin la bandera `g`, el primer elemento de la matriz devuelta es la cadena coincidente, y los elementos restantes son las subcadenas que coinciden con los grupos de captura entre paréntesis de la expresión regular. Así, si `match()` devuelve un array `a`, `a[0]` contiene la coincidencia completa, `a[1]` contiene la subcadena que coincide con la primera expresión entre paréntesis, y así sucesivamente. Para establecer un paralelismo con el método `replace()`, `a[1]` es la misma cadena que `$1`, `a[2]` es la misma que `$2`, y así sucesivamente.

Por ejemplo, considere el análisis de una URL5 con el siguiente código:

```
// Un RegExp de análisis de URL muy sencillo let url = /(\w+):\/\/([\w.]+)\/(\S*)/; let text = "Visita mi blog en http://www.example.com/~david"; let match = text.match(url); let fullurl, protocol, host, path; if (match !== null) { fullurl = match[0]; // fullurl == "http://www.example.com/~david" protocol = match[1]; // protocol == "http" host = match[2]; // host == "www.example.com" path = match[3]; // path == "~david" }
```

En este caso no global, el array devuelto por match() también tiene algunas propiedades de objeto además de los elementos numerados del array. La propiedad input se refiere a la cadena sobre la que se llamó a match(). La propiedad index es la posición dentro de esa cadena en la que comienza la coincidencia. Y si la expresión regular contiene grupos de captura con nombre, entonces el array devuelto también tiene una propiedad groups cuyo valor es un objeto. Las propiedades de este objeto coinciden con los nombres de los grupos nombrados, y los valores son el texto coincidente. Podríamos reescribir el ejemplo anterior de análisis de URL, por ejemplo, así

```
let url = /(?:< protocolo>\w+):\/\/(?:<host>[\w.]+)\/(?: < ruta>\S*)/; let text = "Visita mi blog en http://www.example.com/~david"; let match = text.match(url); match[0] // => "http://www.ejemplo.com/~david" match.input // => text match.index // => 17 match.groups.protocol // => "http" match.groups.host // => "www.example.com" match.groups.path // => "~david"
```

Hemos visto que match() se comporta de forma muy diferente dependiendo de si la RegExp tiene la bandera g activada o no. También hay diferencias importantes pero menos dramáticas en el comportamiento cuando la bandera y está activada. Recuerde que la bandera y hace que una expresión regular sea "pegajosa" al restringir en qué parte de la cadena pueden comenzar las coincidencias. Si una RegExp tiene las banderas g e y activadas, entonces match() devuelve una matriz de cadenas coincidentes, igual que cuando g está activada sin y. Pero la primera coincidencia debe comenzar al principio de la cadena, y cada coincidencia

posterior debe comenzar en el carácter inmediatamente posterior a la coincidencia anterior.

Si la bandera `y` se establece sin `g`, entonces `match()` trata de encontrar una única coincidencia `y`, por defecto, esta coincidencia se limita al inicio de la cadena. Sin embargo, puede cambiar esta posición de inicio de la coincidencia por defecto, estableciendo la propiedad `lastIndex` del objeto `RegExp` en el índice en el que desea encontrar la coincidencia. Si se encuentra una coincidencia, entonces este `lastIndex` se actualizará automáticamente al primer carácter después de la coincidencia, por lo que si llama a `match()` de nuevo, en este caso, buscará una coincidencia posterior. (`lastIndex` puede parecer un nombre extraño para una propiedad que especifica la posición en la que comenzar la *siguiente coincidencia*. Lo veremos de nuevo cuando cubramos el método `RegExp exec()`, y su nombre puede tener más sentido en ese contexto).

```
let vowel = /[aeiou]/y; // Coincidencia de vocales "test". match(vowel) //=> null: "test" no empieza por una vocal. lastIndex = 1; // Especificar una posición de coincidencia diferente "test". match(vowel)[0] //=> "e": encontramos una vocal en la posición 1 vowel. lastIndex //=> 2: lastIndex se actualizó automáticamente "test". match(vowel) //=> null: no hay vocal en la posición 2 vowel. lastIndex //=> 0: lastIndex se pone a cero tras una coincidencia fallida
```

Cabe destacar que pasar una expresión regular no global al método `match()` de una cadena es lo mismo que pasar la cadena al método `exec()` de la expresión regular: la matriz devuelta y sus propiedades son las mismas en ambos casos.

## MATCHALL()

El método matchAll() está definido en ES2020, y desde principios de 2020 está implementado por los navegadores web modernos y por Node. matchAll() espera una RegExp con la bandera g activada. Sin embargo, en lugar de devolver una matriz de subcadenas coincidentes como hace match(), devuelve un iterador que produce el tipo de objetos coincidentes que devuelve match() cuando se utiliza con una RegExp no global. Esto hace que matchAll() sea la forma más fácil y general de recorrer todas las coincidencias dentro de una cadena.

Puede utilizar matchAll() para recorrer las palabras de una cadena de texto de esta manera:

```
// Uno o más caracteres alfabéticos Unicode entre los límites de las palabras
const words = /\b\p{Alphabetic}+\b/gu; // \p no está soportado en Firefox todavía
const text = "Esta es una prueba ingenua del método matchAll().";
for(let word of text.matchAll(words)) {
  console.log(`Found '${word[0]}' at index ${word.index}.`);
}
```

Puede establecer la propiedad lastIndex de un objeto RegExp para decirle a matchAll() en qué índice de la cadena debe empezar a coincidir. Sin embargo, a diferencia de los otros métodos de concordancia de patrones, matchAll() nunca modifica la propiedad lastIndex del RegExp al que se llama, y esto hace que sea mucho menos probable que cause errores en su código.

## SPLIT()

El último de los métodos de expresión regular del objeto String es split(). Este método divide la cadena sobre la que se llama en una matriz de subcadenas, utilizando el argumento como separador. Se puede utilizar con un argumento de cadena como este:

```
"123,456,789". split(",") //=> ["123", "456",  
"789"]
```

El método `split()` también puede tomar una expresión regular como argumento, y esto permite especificar separadores más generales. Aquí lo llamamos con un separador que incluye una cantidad arbitraria de espacios en blanco a cada lado:

```
"1, 2, 3,\n4, 5". split(/\s*,\s*) //=> ["1", "2", "3", "4",  
"5"]
```

Sorprendentemente, si llama a `split()` con un delimitador RegExp y la expresión regular incluye grupos de captura, el texto que coincide con los grupos de captura se incluirá en el array devuelto. Por ejemplo:

```
const htmlTag = /<([^>]+)>/; // < seguido de uno o más no>, seguido de > "Testing<  
br/>1,2,3". split(htmlTag) // => ["Testing", "br/",  
"1,2,3"]
```

### 11.3.3 La clase RegExp

Esta sección documenta el constructor `RegExp()`, las propiedades de las instancias `RegExp`, y dos importantes métodos de concordancia de patrones definidos por la clase `RegExp`.

El constructor `RegExp()` toma uno o dos argumentos de cadena y crea un nuevo objeto `RegExp`. El primer argumento de este constructor es una cadena que contiene el cuerpo de la expresión regular-el texto que aparecería entre las barras en un literal de expresión regular. Tenga en cuenta que tanto los literales de cadena como las expresiones regulares utilizan el carácter `\` para las secuencias de escape, por lo que cuando pase una expresión

regular a RegExp() como un literal de cadena, debe reemplazar cada carácter \\_ por \\_. El segundo argumento de RegExp() es opcional. Si se suministra, indica las banderas de la expresión regular. Debe ser g, i, m, s, u, y, o cualquier combinación de estas letras.

Por ejemplo:

```
// Encuentra todos los números de cinco dígitos en una cadena. Tenga  
// en cuenta el doble  
En este caso. let zipcode = new RegExp("\\d{5}", "g");
```

El constructor RegExp() es útil cuando una expresión regular se crea dinámicamente y, por tanto, no puede representarse con la sintaxis literal de la expresión regular. Por ejemplo, para buscar una cadena introducida por el usuario, se debe crear una expresión regular en tiempo de ejecución con RegExp().

En lugar de pasar una cadena como primer argumento a RegExp(), también puede pasar un objeto RegExp. Esto le permite copiar una expresión regular y cambiar sus banderas:

```
let exactMatch = /JavaScript/;  
let caseInsensitive = new RegExp(exactMatch, "i");
```

## PROPIEDADES REGEXP

Los objetos RegExp tienen las siguientes propiedades:

*fuente*

Esta propiedad de sólo lectura es el texto fuente de la expresión regular: los caracteres que aparecen entre las barras en un literal RegExp.

### *banderas*

Esta propiedad de sólo lectura es una cadena que especifica el conjunto de letras que representan las banderas para la RegExp.

### *global*

Una propiedad booleana de sólo lectura que es verdadera si el indicador g está activado.

### *ignoreCase*

Una propiedad booleana de sólo lectura que es verdadera si el indicador i está activado.

### *multilínea*

Una propiedad booleana de sólo lectura que es verdadera si el indicador m está activado.

### *dotAll*

Una propiedad booleana de sólo lectura que es verdadera si el indicador s está activado.

### *unicode*

Una propiedad booleana de sólo lectura que es verdadera si el indicador u está activado.

### *pegajoso*

Una propiedad booleana de sólo lectura que es verdadera si el indicador y está activado.

### *lastIndex*

Esta propiedad es un entero de lectura/escritura. Para los patrones con las banderas g o y, especifica la posición del carácter en la que

debe comenzar la siguiente búsqueda. Es utilizada por los métodos `exec()` y `test()`, descritos en las dos siguientes subsecciones.

## TEST()

El método `test()` de la clase `RegExp` es la forma más sencilla de utilizar una expresión regular. Toma un único argumento de cadena y devuelve `true` si la cadena coincide con el patrón o `false` si no coincide.

`test()` funciona simplemente llamando al método `exec()` (mucho más complicado) descrito en la siguiente sección y devolviendo `true` si `exec()` devuelve un valor no nulo. Debido a esto, si usa `test()` con un `RegExp` que usa las banderas `g` o `y`, entonces su comportamiento depende del valor de la propiedad `lastIndex` del objeto `RegExp`, que puede cambiar inesperadamente. Vea "[La propiedad lastIndex y la reutilización de RegExp](#)" para más detalles.

## EXEC()

El método `RegExp exec()` es la forma más general y potente de utilizar expresiones regulares. Toma un único argumento de cadena y busca una coincidencia en esa cadena. Si no se encuentra ninguna coincidencia, devuelve `null`. Sin embargo, si se encuentra una coincidencia, devuelve una matriz igual que la devuelta por el método `match()` para búsquedas no globales. El elemento `0` de la matriz contiene la cadena que coincidió con la expresión regular, y cualquier elemento posterior de la matriz contiene las subcadenas que coincidieron con cualquier grupo de captura. El array devuelto también tiene propiedades con nombre: la propiedad `index` contiene la posición del carácter en el que se produjo la

coincidencia, y la propiedad input especifica la cadena en la que se buscó, y la propiedad groups, si está definida, se refiere a un objeto que contiene las subcadenas que coinciden con los grupos de captura que se hayan nombrado.

A diferencia del método String match(), exec() devuelve el mismo tipo de matriz tanto si la expresión regular tiene la bandera global g como si no. Recuerde que match() devuelve una matriz de coincidencias cuando se le pasa una expresión regular global. exec(), por el contrario, siempre devuelve una única coincidencia y proporciona información completa sobre esa coincidencia. Cuando exec() se llama a una expresión regular que tiene la bandera global g o la bandera pegajosa y, consulta la propiedad lastIndex del objeto RegExp para determinar dónde empezar a buscar una coincidencia. (Y si la bandera y está activada, también restringe la coincidencia para que comience en esa posición). Para un objeto RegExp recién creado, lastIndex es 0, y la búsqueda comienza al principio de la cadena. Pero cada vez que exec() encuentra una coincidencia con éxito, actualiza la propiedad lastIndex al índice del carácter inmediatamente posterior al texto coincidente. Si exec() no encuentra una coincidencia, restablece lastIndex a 0. Este comportamiento especial le permite llamar a exec() repetidamente para recorrer todas las coincidencias de expresiones regulares en una cadena. (Aunque, como hemos descrito, en ES2020 y posteriores, el método matchAll() de String es una forma más sencilla de recorrer en bucle todas las coincidencias). Por ejemplo, el bucle del siguiente código se ejecutará dos veces:

```
let pattern = /Java/g; let text = "JavaScript > Java"; let match; while((match = pattern.exec(text)) !== null) { console.log(`Se ha encontrado ${match[0]} en ${match.index}`); console.log(`La siguiente búsqueda comienza en ${pattern.lastIndex}`); }
```

## LA PROPIEDAD LASTINDEX Y LA REUTILIZACIÓN DE REGEXP

Como ya has visto, la API de expresiones regulares de JavaScript es complicada. El uso de la propiedad lastIndex con las banderas g e y es una parte particularmente incómoda de esta API. Cuando se usan estas banderas, hay que tener especial cuidado al llamar a los métodos match(), exec(), o test() porque el comportamiento de estos métodos depende de lastIndex, y el valor de lastIndex depende de lo que se haya hecho previamente con el objeto RegExp. Esto hace que sea fácil escribir código con errores.

Supongamos, por ejemplo, que queremos encontrar el índice de todas las etiquetas <p> dentro de una cadena de texto HTML.

Podríamos escribir un código como éste:

```
let match, positions = [];
while((match = /<p>/g.exec(html)) !== null) { // POSIBLE BUCLE INFINITO
    positions.push(match.index);
}
```

Este código no hace lo que queremos. Si la cadena html contiene al menos una etiqueta <p>, entonces se hará un bucle para siempre. El problema es que usamos un literal RegExp en la condición del bucle while. Para cada iteración del bucle, estamos creando un nuevo objeto RegExp con lastIndex establecido en 0, por lo que exec() siempre comienza en el inicio de la cadena, y si hay una coincidencia, seguirá coincidiendo una y otra vez. La solución, por supuesto, es definir el RegExp una vez, y guardarlo en una variable para que estemos usando el mismo objeto RegExp para cada iteración del bucle.

Por otro lado, a veces la reutilización de un objeto RegExp es un error. Supongamos, por ejemplo, que queremos hacer un bucle a través de todas las palabras de un diccionario para encontrar palabras que contengan pares de letras dobles:

```
let dictionary = ["manzana", "libro", "café"];
let doubleLetterWords = [];
let doubleLetter = /(\w)\1/g;

for(let word of dictionary) {
    if (doubleLetter.test(word)) {
        doubleLetterWords.push(word);
    }
}
doubleLetterWords // => ["manzana", "café"]; ¡"libro" no está!
```

Como hemos puesto la bandera g en la RegExp, la propiedad lastIndex se cambia después de las coincidencias exitosas, y el método test() (que se basa en exec()) comienza a buscar una coincidencia en la posición especificada por lastIndex. Después de coincidir con el "pp" en "apple", lastIndex es 3, y por lo tanto empezamos a buscar la palabra "book" en la posición 3 y no vemos el "oo" que contiene.

Podríamos solucionar este problema eliminando la bandera g (que en realidad no es necesaria en este ejemplo concreto), o moviendo el literal RegExp al cuerpo del bucle para que se vuelva a crear en cada iteración, o poniendo explícitamente a cero lastIndex antes de cada llamada a test().

La moraleja aquí es que lastIndex hace que la API RegExp sea propensa a errores. Así que tenga mucho cuidado cuando use las banderas g o y haga un bucle. Y en ES2020 y posteriores, utilice el método String matchAll()

en lugar de `exec()` para eludir este problema ya que `matchAll()` no modifica `lastIndex`.

## 11.4 Fechas y horarios

La clase Date es la API de JavaScript para trabajar con fechas y horas. Crea un objeto Date con el constructor Date(). Sin argumentos, devuelve un objeto Date que representa la fecha y hora actuales:

```
let now = new Date(); // La hora actual
```

Si pasas un argumento numérico, el constructor Date() interpreta ese argumento como el número de milisegundos desde la época de

1970: `let epoch = new Date(0); // Medianoche, 1 de enero de 1970, GMT`

Si especifica dos o más argumentos enteros, se interpretan como el año, el mes, el día del mes, la hora, los minutos, los segundos y los milisegundos en su zona horaria local, como en lo siguiente:

```
let century = new Date(2100, // Año 2100 0, // Enero  
1, // 1er.  
2, 3, 4, 5); // 02:03:04.005, hora local
```

Una peculiaridad de la API de fechas es que el primer mes de un año es el número 0, pero el primer día de un mes es el número 1. Si se omiten los campos de hora, el constructor de Date() los pone por defecto a 0, estableciendo la hora a medianoche.

Tenga en cuenta que cuando se invoca con varios números, el constructor de Date() los interpreta utilizando la zona horaria a la

que esté configurado el ordenador local. Si quiere especificar una fecha y hora en UTC (Tiempo Universal Coordinado, también conocido como GMT), entonces puede utilizar el método Date.UTC(). Este método estático toma los mismos argumentos que el constructor Date(), los interpreta en UTC y devuelve una marca de tiempo en milisegundos que puede pasar al constructor Date():

```
// Medianoche en Inglaterra, 1 de enero de 2100 let century =  
new Date(Date.UTC(2100, 0, 1));
```

Si imprime una fecha (con console.log(century), por ejemplo), se imprimirá, por defecto, en su zona horaria local. Si quiere mostrar una fecha en UTC, debe convertirla explícitamente en una cadena con toUTCString() o toISOString().

Por último, si se pasa una cadena al constructor de Date(), éste intentará analizar esa cadena como una especificación de fecha y hora. El constructor puede analizar fechas especificadas en los formatos producidos por los métodos toString(), toUTCString() y toISOString():

```
let century = new Date("2100-01-01T00:00:00Z"); // Una fecha con formato ISO
```

Una vez que se tiene un objeto Date, varios métodos get y set permiten consultar y modificar los campos año, mes, día del mes, hora, minuto, segundo y milisegundo de la Date. Cada uno de estos métodos tiene dos formas: una que obtiene o establece utilizando la hora local y otra que obtiene o establece utilizando la hora UTC. Para obtener o establecer el año de un objeto Date, por ejemplo

por ejemplo, utilizaría `getFullYear()`, `getUTCFullYear()`, `setFullYear()` o `setUTCFullYear()`:

```
let d = new Date(); // Comienza con la fecha actual  
d. setFullYear(d. getFullYear() + 1); // Incrementar el año
```

Para obtener o establecer los demás campos de una fecha, sustituya "FullYear" en el nombre del método por "Month", "Date", "Hours", "Minutes", "Seconds" o "Milliseconds". Algunos de los métodos de establecimiento de fechas permiten establecer más de un campo a la vez. `setFullYear()` y `setUTCFullYear()` también permiten, opcionalmente, establecer el mes y el día del mes. Y `setHours()` y `setUTCHours()` permiten especificar los campos de minutos, segundos y milisegundos, además del campo de horas.

Tenga en cuenta que los métodos para consultar el día del mes son `getDate()` y `getUTCDate()`. Las funciones `getDay()` y `getUTCDay()`, más naturales, devuelven el día de la semana (de 0 para el domingo a 6 para el sábado). El día de la semana es de sólo lectura, por lo que no hay un método `setDay()` correspondiente.

### 11.4.1 Marcas de tiempo

JavaScript representa las fechas internamente como enteros que especifican el número de milisegundos desde (o antes de) la medianoche del 1 de enero de 1970, hora UTC. Se admiten enteros de hasta 8.640.000.000.000, por lo que JavaScript no se quedará sin milisegundos durante más de 270.000 años.

Para cualquier objeto Date, el método `getTime()` devuelve este valor interno, y el método  `setTime()` lo establece. Así que puedes

añadir 30 segundos a un Date con un código como este, por ejemplo:

```
d.setTime(d.getTime() + 30000);
```

Estos valores en milisegundos se llaman a veces *timestamps*, y a veces es útil trabajar con ellos directamente en lugar de con objetos Date. El método estático Date.now() devuelve la hora actual como una marca de tiempo y es útil cuando se quiere medir el tiempo que tarda el código en ejecutarse:

## MARCAS DE TIEMPO DE ALTA RESOLUCIÓN

Las marcas de tiempo devueltas por Date.now() se miden en milisegundos. Un milisegundo es, en realidad, un tiempo relativamente largo para un ordenador, y a veces se puede querer medir el tiempo transcurrido con mayor precisión. La función performance.now() lo permite: también devuelve una marca de tiempo basada en milisegundos, pero el valor devuelto no es un entero, por lo que incluye fracciones de milisegundo. El valor devuelto por performance.now() no es una marca de tiempo absoluta como lo es el valor de Date.now(). En su lugar, simplemente indica cuánto tiempo ha transcurrido desde que se cargó una página web o desde que se inició el proceso de Node.

El objeto performance forma parte de una API de rendimiento más amplia que no está definida por el estándar ECMAScript pero que es implementada por los navegadores web y por Node. Para utilizar el objeto performance en Node, debes importarlo con:

```
const { rendimiento } = require("perf_hooks");
```

Permitir un cronometraje de alta precisión en la web puede permitir que sitios web sin escrúpulos tomen huellas digitales de los visitantes, por lo que los navegadores (especialmente Firefox) pueden reducir la precisión de performance.now() por defecto. Como desarrollador web, deberías poder volver a habilitar la temporización de alta precisión de alguna manera (como por ejemplo, estableciendo privacy.reduceTimerPrecision en false en Firefox).

```
let startTime = Date.now(); reticulateSplines(); // Hacer alguna operación que
consume tiempo let endTime = Date.now(); console.log(`La reticulación de la
spline tomó ${endTime - startTime}ms.');
```

## 11.4.2 Aritmética de fechas

Los objetos Date se pueden comparar con los operadores de comparación estándar de JavaScript <, <=, > y >=. Y puedes restar un objeto Date de otro para determinar el número de milisegundos entre las dos fechas. (Esto funciona porque la clase Date define un método valueOf() que devuelve una marca de tiempo).

Si quiere añadir o restar un número determinado de segundos, minutos u horas a una fecha, lo más fácil es modificar simplemente la marca de tiempo, como se demostró en el ejemplo anterior, cuando añadimos 30 segundos a una fecha. Esta técnica se vuelve más engorrosa si quiere añadir días, y no funciona en absoluto para los meses y los años, ya que tienen un número variable de días. Para



DeepL

Suscríbete a DeepL Pro para poder editar este documento.  
Entra en [www.DeepL.com/pro](https://www.DeepL.com/pro) para más información.

para hacer aritmética de fechas con días, meses y años, puede utilizar setDate(), setMonth() y setYear(). Aquí, por ejemplo, hay un código que añade tres meses y dos semanas a la fecha actual:

```
let d = new Date();
d.setMonth(d.getMonth() + 3, d.getDate() + 14);
```

Los métodos de fijación de fecha funcionan correctamente incluso cuando se desbordan. Cuando añadimos tres meses al mes actual, podemos acabar con un valor mayor que 11 (que representa diciembre). La función setMonth() maneja esto incrementando el año según sea necesario. Del mismo modo, cuando establecemos el día del mes a un valor mayor que el número de días del mes, el mes se incrementa adecuadamente.

#### 11.4.3. Formateo y análisis de cadenas de fechas

Si utiliza la clase Date para llevar la cuenta de las fechas y horas (a diferencia de la medición de intervalos de tiempo), entonces es probable que necesite mostrar fechas y horas a los usuarios de su código. La clase Date define una serie de métodos diferentes para convertir los objetos Date en cadenas. He aquí algunos ejemplos:

```
let d = new Date(2020, 0, 1, 17, 10, 30); // 17:10:30 del día de Año Nuevo de 2020
d.toString() // => "Wed Jan 01 2020 17:10:30 GMT-0800 (Pacific Standard Time)"
d.toUTCString() // => "Thu, 02 Jan 2020 01:10:30 GMT"
d.toLocaleDateString() // => "1/1/2020": 'en-US' locale
d.toLocaleTimeString() // => "5:10:30 PM": 'en-US' locale
d.toISOString() // => "2020-01-02T01:10:30.000Z"
```

Esta es una lista completa de los métodos de formato de cadena de la clase Date:

*toString()*

Este método utiliza la zona horaria local, pero no formatea la fecha y la hora de manera local.

#### *toUTCString()*

Este método utiliza la zona horaria UTC, pero no formatea la fecha de manera local.

#### *toISOString()*

Este método imprime la fecha y la hora en el formato estándar año-mes-día horas:minutos:segundos.ms de la norma ISO-8601. La letra "T" separa la parte de la fecha de la parte de la hora. La hora se expresa en UTC, y esto se indica con la letra "Z" como última letra de la salida.

#### *toLocaleString()*

Este método utiliza la zona horaria local y un formato apropiado para la localidad del usuario.

#### *toString()*

Este método sólo formatea la parte de la fecha y omite la hora. Utiliza la zona horaria local y no hace el formato apropiado para la localidad.

#### *toLocaleDateString()*

Este método sólo formatea la fecha. Utiliza la zona horaria local y un formato de fecha apropiado para la localidad.

#### *toTimeString()*

Este método sólo formatea la hora y omite la fecha. Utiliza la zona horaria local, pero no formatea la hora de manera local.

#### *toLocaleTimeString()*

Este método formatea la hora de manera local y utiliza la zona horaria local.

Ninguno de estos métodos de conversión de fecha en cadena es ideal para formatear las fechas y horas que se mostrarán a los usuarios finales. Véase en [§11.7.2](#) una técnica de formateo de fecha y hora más general y que tiene en cuenta la localización.

Por último, además de estos métodos que convierten un objeto Date en una cadena, también existe un método estático Date.parse() que toma una cadena como argumento, intenta analizarla como fecha y hora, y devuelve una marca de tiempo que representa esa fecha. Date.parse() puede analizar las mismas cadenas que el constructor Date() y se garantiza que puede analizar la salida de toISOString(), toUTCString() y toString().

## 11.5 Clases de error

Las sentencias throw y catch de JavaScript pueden lanzar y atrapar cualquier valor de JavaScript, incluidos los valores primitivos. No hay ningún tipo de excepción que deba utilizarse para señalar errores. Sin embargo, JavaScript define una clase Error, y es tradicional utilizar instancias de Error o una subclase cuando se señala un error con throw. Una buena razón para utilizar un objeto Error es que, cuando se crea un Error, éste captura el estado de la pila de JavaScript, y si la excepción no es capturada, el seguimiento de la pila se mostrará con el mensaje de error, lo que le ayudará a depurar el problema. (Tenga en cuenta que el seguimiento de la pila muestra dónde se creó el objeto Error, no dónde lo lanza la sentencia throw. Si siempre crea el objeto justo antes de lanzarlo con throw new Error(), esto no causará ninguna confusión).

Los objetos Error tienen dos propiedades: message y name, y un método toString(). El valor de la propiedad message es el valor que

se ha pasado al constructor de `Error()`, convertido a una cadena si es necesario. Para los objetos de error creados con `Error()`, la propiedad `name` es siempre "Error". El método `toString()` simplemente devuelve el valor de la propiedad `name` seguido de dos puntos y un espacio y el valor de la propiedad `message`.

Aunque no forma parte del estándar ECMAScript, Node y todos los navegadores modernos también definen una propiedad de pila en los objetos `Error`. El valor de esta propiedad es una cadena de varias líneas que contiene un seguimiento de la pila de llamadas de JavaScript en el momento en que se creó el objeto `Error`. Esto puede ser una información útil para registrar cuando se detecta un error inesperado.

Además de la clase `Error`, JavaScript define una serie de subclases que utiliza para señalar determinados tipos de errores definidos por ECMAScript. Estas subclases son `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError` y `URIError`. Puede utilizar estas clases de error en su propio código si le parecen apropiadas. Al igual que la clase `Error` base, cada una de estas subclases tiene un constructor que toma un único argumento de mensaje. Y las instancias de cada una de estas subclases tienen una propiedad `name` cuyo valor es el mismo que el nombre del constructor.

Debería sentirse libre de definir sus propias subclases de `Error` que mejor encapsulen las condiciones de error de su propio programa. Tenga en cuenta que no está limitado a las propiedades de nombre y mensaje. Si crea una subclase, puede definir nuevas propiedades para proporcionar detalles del error. Si está escribiendo un analizador sintáctico, por ejemplo, podría ser útil definir una clase

ParseError con propiedades de línea y columna que especifiquen la ubicación exacta del fallo de análisis. O si estás trabajando con peticiones HTTP, puede que quieras definir una clase HTTPError que tenga una propiedad de estado que contenga el código de estado HTTP (como 404 o 500) de la petición fallida.

Por ejemplo:

```
class HTTPError extends Error { constructor(status, statusText, url)  
{ super(`${status} ${statusText}: ${url}`); this. status = status; this.  
statusText = statusText;  
  
this. url = url; }  
  
get name() { return "HTTPError"; } }  
  
let error = new HTTPError(404, "Not Found",  
"http://example.com/"); error. status // => 404 error.  
message // => "404 Not Found:  
http://example.com/" error. name // => "HTTPError"
```

## 11.6 Serialización y análisis de JSON

Cuando un programa necesita guardar datos o necesita transmitir datos a través de una conexión de red a otro programa, debe convertir sus estructuras de datos en memoria en una cadena de bytes o caracteres que puedan ser guardados o transmitidos y luego ser analizados para restaurar las estructuras de datos originales en memoria. Este proceso de convertir las estructuras de datos en flujos de bytes o caracteres se conoce como *serialización* (o *marshaling* o incluso *pickling*).

La forma más sencilla de serializar datos en JavaScript utiliza un formato de serialización conocido como JSON. Este acrónimo

significa "JavaScript Object Notation" y, como su nombre indica, el formato utiliza la sintaxis literal de objetos y matrices de JavaScript para convertir las estructuras de datos formadas por objetos y matrices en cadenas. JSON admite números y cadenas primitivas y también los valores true, false y null, así como matrices y objetos construidos a partir de esos valores primitivos. JSON no admite otros tipos de JavaScript como Map, Set, RegExp, Date o arrays tipados. Sin embargo, ha demostrado ser un formato de datos notablemente versátil y es de uso común incluso en programas no basados en JavaScript.

JavaScript soporta la serialización y deserialización de JSON con las dos funciones `JSON.stringify()` y `JSON.parse()`, que fueron cubiertas brevemente en §6.8. Dado un objeto o array (anidado a una profundidad arbitraria) que no contenga ningún valor no serializable como objetos `RegExp` o arrays tipados, puedes serializar el objeto simplemente pasándolo a `JSON.stringify()`. Como su nombre indica, el valor de retorno de esta función es una cadena. Y dada una cadena devuelta por `JSON.stringify()`, puedes recrear la estructura de datos original pasando la cadena a `JSON.parse()`:

```
let o = {s: "", n: 0, a: [true, false, null]}; let s = JSON.stringify(o); // s ==
'{"s": "", "n":0, "a": [true,false,null]}'
let copy = JSON.parse(s); // copy == {s:
"", n: 0, a: [true, false, null]}
```

Si dejamos de lado la parte en la que los datos serializados se guardan en un archivo o se envían a través de la red, podemos utilizar este par de funciones como una forma algo ineficiente de crear una copia profunda de un objeto:

```
// Hacer una copia profunda de cualquier objeto o matriz serializable
function deepcopy(o) { return JSON.parse(JSON.stringify(o)); }
```

## JSON ES UN SUBCONJUNTO DE JAVASCRIPT

Cuando los datos se serializan en formato JSON, el resultado es un código fuente JavaScript válido para una expresión que se evalúa como una copia de la estructura de datos original. Si prefijas una cadena JSON con var data = y pasas el resultado a eval(), obtendrás una copia de la estructura de datos original asignada a la variable data. Sin embargo, nunca deberías hacer esto, porque es un enorme agujero de seguridad-si un atacante pudiera injectar código JavaScript arbitrario en un archivo JSON, podría hacer que tu programa ejecutara su código. Es más rápido y seguro utilizar simplemente JSON.parse() para decodificar los datos con formato JSON.

JSON se utiliza a veces como un formato de archivo de configuración legible por humanos. Si se encuentra editando a mano un archivo JSON, tenga en cuenta que el formato JSON es un subconjunto muy estricto de JavaScript. Los comentarios no están permitidos y los nombres de las propiedades deben ir entre comillas dobles incluso cuando JavaScript no lo requiera.

Normalmente, se pasa un solo argumento a JSON.stringify() y JSON.parse(). Ambas funciones aceptan un segundo argumento opcional que nos permite extender el formato JSON, y que se describe a continuación. JSON.stringify() también toma un tercer argumento opcional que discutiremos primero. Si quieres que tu cadena con formato JSON sea legible para los humanos (si se está utilizando como un archivo de configuración, por ejemplo), entonces debes pasar null como segundo argumento y pasar un número o una cadena como tercer argumento. Este tercer argumento indica a JSON.stringify() que debe formatear los datos en varias líneas con sangría. Si el tercer argumento es un número, entonces utilizará ese número de espacios para cada nivel de

sangría. Si el tercer argumento es una cadena de espacios en blanco (como '\t'), utilizará esa cadena para cada nivel de sangría.

```
let o = {s: "test", n: 0};  
JSON.stringify(o, null, 2) //=> '{\n "s": "test",\n "n":  
0\n}'
```

JSON.parse() ignora los espacios en blanco, por lo que pasar un tercer argumento a

JSON.stringify() no tiene ningún impacto en nuestra capacidad de convertir la cadena de nuevo en una estructura de datos.

### 11.6.1 Personalizaciones JSON

Si se le pide a JSON.stringify() que serialice un valor que no está soportado de forma nativa por el formato JSON, busca si ese valor tiene un método toJSON(), y si es así, llama a ese método y luego encadena el valor devuelto en lugar del valor original. Los objetos Date implementan toJSON(): devuelve la misma cadena que el método toISOString(). Esto significa que si serializas un objeto que incluye una Fecha, la fecha se convertirá automáticamente en una cadena para ti. Cuando analices la cadena serializada, la estructura de datos recreada no será exactamente la misma que con la que empezaste porque tendrá una cadena donde el objeto original tenía una Fecha.

Si necesita volver a crear objetos Date (o modificar el objeto analizado de cualquier otra forma), puede pasar una función "reviver" como segundo argumento a JSON.parse(). Si se especifica, esta función "reviver" se invoca una vez por cada valor primitivo (pero no los objetos o matrices que contienen esos valores primitivos) analizado desde la cadena de entrada. La

función se invoca con dos argumentos. El primero es un nombre de propiedad, ya sea un nombre de propiedad de objeto o un índice de matriz convertido a una cadena. El segundo argumento es el valor primitivo de esa propiedad de objeto o elemento de matriz. Además, la función se invoca como un método del objeto o matriz que contiene el valor primitivo, por lo que puede referirse a ese objeto contenedor con la palabra clave this.

El valor de retorno de la función reviver se convierte en el nuevo valor de la propiedad nombrada. Si devuelve su segundo argumento, la propiedad permanecerá sin cambios. Si devuelve undefined, entonces la propiedad nombrada será eliminada del objeto o array antes de que JSON.parse() vuelva al usuario.

Como ejemplo, he aquí una llamada a JSON.parse() que utiliza una función reviver para filtrar algunas propiedades y recrear objetos Date:

```
let data = JSON.parse(text, function(key, value) {
    // Eliminar cualquier valor cuyo nombre de propiedad comience con un guión bajo
    if (key[0] === "_") return undefined;

    // Si el valor es una cadena en formato de fecha ISO 8601 conviértalo en una Fecha.
    if (typeof value === "string" && /^[\d\d\dT\d\d\d:\d\d:\d\d.\d\d\d$]/.test(value)) { return new
        Date(value); }

    // En caso contrario, devuelve el valor sin modificar el valor de
    retorno;
});
```

Además del uso de toJSON() descrito anteriormente, JSON.stringify() también permite personalizar su salida pasando una matriz o una función como segundo argumento opcional.

Si en lugar de ello se pasa una matriz de cadenas (o números -se convierten en cadenas-) como segundo argumento, éstas se utilizan como nombres de las propiedades del objeto (o elementos de la matriz). Cualquier propiedad cuyo nombre no esté en la matriz se omitirá en la encadenación. Además, la cadena devuelta incluirá las propiedades en el mismo orden en que aparecen en la matriz (lo que puede ser muy útil al escribir pruebas).

Si se pasa una función, se trata de una función de reemplazo, es decir, la inversa de la función opcional de reenvío que se puede pasar a JSON.parse(). Si se especifica, la función de sustitución se invoca para cada valor que se va a encadenar. El primer argumento de la función replacer es el nombre de la propiedad del objeto o el índice de la matriz del valor dentro de ese objeto, y el segundo argumento es el propio valor. La función de sustitución se invoca como un método del objeto o matriz que contiene el valor que se va a encadenar. El valor devuelto por la función de sustitución se encadena en lugar del valor original. Si el sustituto devuelve algo indefinido o no devuelve nada, ese valor (y su elemento de matriz o propiedad de objeto) se omite en la encadenación.

```
// Especifica qué campos serializar y en qué orden hacerlo let text = JSON.stringify(address, ["city", "state", "country"]);

// Especificar una función de sustitución que omita las propiedades de los valores
RegExp let json = JSON.stringify(o, (k, v) => v instanceof RegExp ?
  indefinido : v);
```

Las dos llamadas a JSON.stringify() aquí utilizan el segundo argumento de manera benigna, produciendo una salida serializada que puede ser deserializada sin requerir una función especial de reviver. En general, sin embargo, si defines un método toJSON() para un tipo, o si utilizas una función de reemplazo que realmente

reemplaza los valores no serializables con los serializables, entonces normalmente necesitarás utilizar una función reviver personalizada con `JSON.parse()` para recuperar tu estructura de datos original. Si haces esto, debes entender que estás definiendo un formato de datos personalizado y sacrificando la portabilidad y compatibilidad con un gran ecosistema de herramientas y lenguajes compatibles con JSON.

## 11.7 La API de internacionalización

La API de internacionalización de JavaScript consta de las tres clases `Intl.NumberFormat`, `Intl.DateTimeFormat` e `Intl.Collator`, que nos permiten dar formato a los números (incluidas las cantidades monetarias y los porcentajes), a las fechas y a las horas de forma adecuada a la localización, así como comparar cadenas de caracteres de forma adecuada a la localización. Estas clases no forman parte del estándar ECMAScript, pero están definidas como parte del [estándar ECMA402](#) y están bien soportadas por los navegadores web. La API `Intl` también está soportada en Node, pero en el momento de escribir este artículo, los binarios de Node preconstruidos no incluyen los datos de localización necesarios para que funcionen con otras localizaciones distintas al inglés de EE.UU.. Por lo tanto, para utilizar estas clases con Node, es posible que tenga que descargar un paquete de datos separado o utilizar una construcción personalizada de Node.

Una de las partes más importantes de la internacionalización es mostrar el texto traducido al idioma del usuario. Hay varias formas de conseguirlo, pero ninguna de ellas está dentro del ámbito de la API `Intl` descrita aquí.

### 11.7.1 Formato de los números

Los usuarios de todo el mundo esperan que los números tengan un formato diferente. Los puntos decimales pueden ser puntos o comas. Los separadores de miles pueden ser comas o puntos, y no se utilizan cada tres dígitos en todos los lugares. Algunas monedas se dividen en centésimas, otras en milésimas y otras no tienen subdivisiones. Por último, aunque los llamados "números arábigos" del 0 al 9 se utilizan en muchos idiomas, esto no es universal, y los usuarios de algunos países esperarán ver los números escritos con los dígitos de sus propias escrituras.

La clase `Intl.NumberFormat` define un método `format()` que tiene en cuenta todas estas posibilidades de formato. El constructor toma dos argumentos. El primer argumento especifica la configuración regional para la que debe formatearse el número y el segundo es un objeto que especifica más detalles sobre cómo debe formatearse el número. Si el primer argumento se omite o no se define, se utilizará la configuración regional del sistema (que suponemos que es la preferida por el usuario). Si el primer argumento es una cadena, especifica la configuración regional deseada, como "en-US" (inglés utilizado en Estados Unidos), "fr" (francés) o "zh-Hans-CN" (chino, utilizando el sistema de escritura Han simplificado, en China). El primer argumento también puede ser una matriz de cadenas de localización, y en este caso, `Intl.NumberFormat` elegirá el más específico que esté bien soportado.

El segundo argumento del constructor `Intl.NumberFormat()`, si se especifica, debe ser un objeto que defina una o más de las siguientes propiedades:

### *estilo*

Especifica el tipo de formato de los números que se requiere. El valor por defecto es "decimal". Especifique "porcentaje" para formatear un número como un porcentaje o especifique "moneda" para especificar un número como una cantidad de dinero.

### *moneda*

Si el estilo es "moneda", entonces esta propiedad es necesaria para especificar el código de moneda ISO de tres letras (como "USD" para dólares estadounidenses o "GBP" para libras esterlinas) de la moneda deseada.

### *currencyDisplay*

Si el estilo es "moneda", esta propiedad especifica cómo se muestra la moneda. El valor por defecto "símbolo" utiliza un símbolo de moneda si la moneda tiene uno. El valor "code" utiliza el código ISO de tres letras, y el valor "name" deletrea el nombre de la moneda en forma larga.

### *useGrouping*

Establezca esta propiedad en false si no desea que los números tengan separadores de miles (o sus equivalentes adaptados a la localidad).

### *minimumIntegerDigits*

El número mínimo de dígitos a utilizar para mostrar la parte entera del número. Si el número tiene menos dígitos que esto, se llenará a la izquierda con ceros. El valor por defecto es 1, pero se pueden utilizar valores hasta 21.

### *mínimoFractionDigits, máximoFractionDigits*

Estas dos propiedades controlan el formato de la parte fraccionaria del número. Si un número tiene menos dígitos fraccionarios que el mínimo, se llenará con ceros a la derecha. Si tiene más que el máximo, la parte fraccionaria se redondeará. Los valores legales para ambas propiedades están entre 0 y 20. El mínimo por defecto es 0 y el máximo por defecto es 3, excepto cuando se formatean importes monetarios, en los que la longitud de la parte fraccionaria varía en función de la moneda especificada.

*minimumSignificantDigits, maximumSignificantDigits* Estas propiedades controlan el número de dígitos significativos utilizados cuando se formatea un número, por lo que son adecuadas para formatear datos científicos, por ejemplo. Si se especifican, estas propiedades anulan las propiedades de dígitos enteros y fraccionarios enumeradas anteriormente. Los valores legales están entre 1 y 21.

Una vez creado un objeto Intl.NumberFormat con la configuración regional y las opciones deseadas, se utiliza pasando un número a su método format(), que devuelve una cadena con el formato adecuado. Por ejemplo:

```
let euros = Intl. NumberFormat("es", {style: "currency", currency: "EUR"});  
euros.format(10) //=> "10,00 €": diez euros, formato español
```

```
let pounds = Intl. NumberFormat("es", {style: "currency", currency: "GBP"});  
libras.format(1000) //=> "£1,000.00": Mil libras,  
Formato en inglés
```

Una característica útil de Intl.NumberFormat (y también de las otras clases Intl) es que su método format() está ligado al objeto NumberFormat al que pertenece. Así, en lugar de definir una variable que haga referencia al objeto de formato y luego invocar el método format() sobre ella, puede simplemente asignar el

método format() a una variable y utilizarlo como si fuera una función independiente, como en este ejemplo:

```
deje los datos = [0,05, . 75, 1];
let formatData = Intl. NumberFormat(undefined, { style: "porcentaje",
minimumFractionDigits: 1, maximumFractionDigits: 1 }). format; data.
map(formatData) // => ["5.0%", "75.0%", "100.0%"]: en la configuración regional en-US
```

Algunos idiomas, como el árabe, utilizan su propia escritura para los dígitos decimales:

```
let arabic = Intl. NumberFormat("ar", {useGrouping:
false}). format; arabic(1234567890) // =>
"١٢٣٤٥٦٧٨٩٠"
```

Otros idiomas, como el hindi, utilizan una escritura que tiene su propio conjunto de dígitos, pero tienden a utilizar los dígitos ASCII 0-9 por defecto. Si desea anular la escritura por defecto utilizada para los dígitos, añada -u-nu- a la configuración regional y sígala con un nombre de escritura abreviado. Puede formatear los números con agrupación de estilo indio y dígitos devanagari de esta manera, por ejemplo:

```
let hindi = Intl. NumberFormat("hi-IN-u-nu-deva"). format; hindi(1234567890) // =>
"੧,੨੩,੪੫,੬੭,੮੯੦"
```

nu es el nombre de la extensión del sistema de numeración, y deva es la abreviatura de Devanagari. La norma Intl API define nombres para otros sistemas de numeración, principalmente para las lenguas índicas del sur y el sudeste de Asia.

## 11.7.2 Formato de fechas y horas

La clase Intl.DateTimeFormat es muy parecida a la clase Intl.NumberFormat. El constructor de Intl.DateTimeFormat() toma

los mismos dos argumentos que Intl.NumberFormat(): una configuración regional o matriz de configuraciones regionales y un objeto de opciones de formato. Y la forma de utilizar una instancia de Intl.DateTimeFormat es llamando a su método format() para convertir un objeto Date en una cadena.

Como se mencionó en el apartado [11.4](#), la clase Date define las clases simples

toLocaleDateString() y toLocaleTimeString() que producen una salida adecuada a la localización del usuario. Pero estos métodos no le dan ningún control sobre los campos de la fecha y la hora que se muestran. Tal vez quiera omitir el año pero añadir un día de la semana al formato de la fecha. ¿Quiere que el mes se represente numéricamente o se deletree por su nombre? La clase Intl.DateTimeFormat proporciona un control detallado sobre lo que se muestra basándose en las propiedades del objeto de opciones que se pasa como segundo argumento al constructor.

Tenga en cuenta, sin embargo, que Intl.DateTimeFormat no siempre puede mostrar exactamente lo que usted pide. Si especifica las opciones para formatear las horas y los segundos pero omite los minutos, encontrará que el formateador muestra los minutos de todos modos. La idea es que utilices el objeto de opciones para especificar qué campos de fecha y hora quieres presentar al usuario y cómo quieras que se formateen (por nombre o por número, por ejemplo), entonces el formateador buscará un formato apropiado para la localidad que más se acerque a lo que has pedido.

Las opciones disponibles son las siguientes. Especifique sólo las propiedades de los campos de fecha y hora que desee que aparezcan en la salida formateada.

#### *año*

Utilice "numérico" para un año completo de cuatro dígitos o "2 dígitos" para una abreviatura de dos dígitos.

#### *mes*

Utilice "numérico" para un número posiblemente corto como "1", o "2digit" para una representación numérica que siempre tiene dos dígitos, como "01". Utilice "long" para un nombre completo como "January", "short" para un nombre abreviado como "Jan", y "narrow" para un nombre muy abreviado como "J" que no se garantiza que sea único.

#### *día*

Utilice "numérico" para un número de uno o dos dígitos o "2 dígitos" para un número de dos dígitos para el día del mes.

#### *entre semana*

Utilice "largo" para un nombre completo como "Lunes", "corto" para un nombre abreviado como "Mon", y "estrecho" para un nombre muy abreviado como "M" que no se garantiza que sea único.

#### *era*

Esta propiedad especifica si una fecha debe ser formateada con una era, como CE o BCE. Esto puede ser útil si está formateando fechas de hace mucho tiempo o si está usando un calendario japonés. Los valores legales son "largo", "corto" y "estrecho".

### *hora, minuto, segundo*

Estas propiedades especifican cómo desea que se muestre la hora. Utilice "numérico" para un campo de uno o dos dígitos o "2 dígitos" para forzar que los números de un solo dígito se rellenen a la izquierda con un 0.

### *zona horaria*

Esta propiedad especifica la zona horaria deseada para la que se debe formatear la fecha. Si se omite, se utiliza la zona horaria local. Las implementaciones siempre reconocen "UTC" y también pueden reconocer Autoridad de Asignación de Números de Internet (IANA): nombres de zonas horarias,

como "América/Los\_Angeles".

### *timeZoneName*

Esta propiedad especifica cómo debe mostrarse la zona horaria en una fecha u hora con formato. Utilice "long" para un nombre de zona horaria completamente deletreado y "short" para una zona horaria abreviada o numérica.

### *hora12*

Esta propiedad booleana especifica si se utiliza o no la hora de 12 horas. El valor por defecto depende de la localización, pero se puede anular con esta propiedad.

### *hourCycle*

Esta propiedad permite especificar si la medianoche se escribe como 0 horas, 12 horas o 24 horas. El valor por defecto depende de la localización, pero puede anularlo con esta propiedad. Tenga en cuenta que la hora12 tiene prioridad sobre esta propiedad. Utilice el valor "h11" para especificar que la medianoche es 0 y la hora anterior a la medianoche es

11pm. Utilice "h12" para especificar que la medianoche es 12. Utilice "h23" para especificar que la medianoche es 0 y la hora anterior a la medianoche es 23. Y utilice "h24" para especificar que la medianoche es 24.

He aquí algunos ejemplos:

```
let d = new Date("2020-01-02T13:14:15Z"); // 2 de enero de 2020, 13:14:15 UTC

// Sin opciones, obtenemos un formato de fecha numérico básico
Intl.DateTimeFormat("en-US").format(d) //=> "1/2/2020"
Intl.DateTimeFormat("fr-FR").format(d) //=> "02/01/2020"

// El día de la semana y el mes deletreados let opts = { día de la semana: "largo", mes:
// "largo", año: "numérico", día: "numérico" };
Intl.DateTimeFormat("en-US", opts).format(d) //=> "Jueves, 2 de enero de 2020" Intl.DateTimeFormat("es-ES", opts).format(d) //=> "jueves, 2 de enero de 2020"

// La hora en Nueva York, para un canadiense francófono opts = { hour: "numérico",
// minuto: "2-digit", timeZone: "America/New_York" };
Intl.DateTimeFormat("fr-CA", opts).format(d) //=> "8 h 14"
```

Intl.DateTimeFormat puede mostrar fechas utilizando calendarios distintos al calendario juliano por defecto basado en la era cristiana. Aunque algunas localizaciones pueden utilizar un calendario no cristiano por defecto, siempre se puede especificar explícitamente el calendario a utilizar añadiendo -u-ca- a la localización y siguiendo con el nombre del calendario. Los posibles nombres de los calendarios son "budista", "chino", "copto", "etíope", "gregoriano", "hebreo", "indio", "islámico", "iso8601", "japonés" y "persa". Continuando con el ejemplo anterior, podemos determinar el año en varios calendarios no cristianos:

```
let opts = { año: "numérico", era: "short" };
```

```
Intl.DateTimeFormat("es", opts).format(d) //  
=> "2020 AD" Intl.DateTimeFormat("en-u-ca-iso8601", opts).format(d) //  
=> "2020 AD" Intl.DateTimeFormat("en-u-ca-hebreo", opts).format(d) //  
=> "5780 AM" Intl.DateTimeFormat("en-u-ca-buddhist", opts).format(d) //  
=> "2563 BE" Intl.DateTimeFormat("en-u-ca-islamic", opts).format(d) //  
=> "1441 AH" Intl.DateTimeFormat("en-u-ca-persa", opts).format(d) //  
=> "1398 AP" Intl.DateTimeFormat("en-u-ca-indian", opts).format(d) //  
=> "1941 Saka" Intl.DateTimeFormat("en-u-ca-chinese", opts).format(d) //  
=> "36 78" Intl.DateTimeFormat("en-u-ca-japonés", opts).format(d) //  
=> "2 Reiwa"
```

### 11.7.3 Comparación de cadenas

El problema de ordenar las cadenas en orden alfabético (o algún "orden de cotejo" más general para las escrituras no alfabéticas) es más desafiante de lo que los angloparlantes a menudo se dan cuenta. El inglés utiliza un alfabeto relativamente pequeño, sin letras acentuadas, y tenemos la ventaja de contar con una codificación de caracteres (ASCII, desde que se incorporó a Unicode) cuyos valores numéricos se ajustan perfectamente a nuestro orden de clasificación de cadenas estándar. Las cosas no son tan sencillas en otros idiomas. El español, por ejemplo, trata la ñ como una letra distinta que viene después de la n y antes de la o. El lituano alfabetiza la Y antes de la J, y el galés trata los dígrafos como la CH y la DD como letras únicas, con la CH después de la C y la DD después de la D.

Si quiere mostrar cadenas a un usuario en un orden que le resulte natural, no es suficiente con utilizar el método `sort()` sobre un array de cadenas. Pero si se crea un objeto `Intl.Collator`, se puede pasar el método `compare()` de ese objeto al método `sort()` para realizar una ordenación localmente apropiada de las cadenas. Los objetos `Intl.Collator` pueden configurarse para que el método `compare()` realice comparaciones que no distingan entre

mayúsculas y minúsculas o incluso comparaciones que sólo consideren la letra base e ignoren los acentos y otros signos diacríticos.

Al igual que Intl.NumberFormat() e Intl.DateTimeFormat(), el constructor de Intl.Collator() toma dos argumentos. El primero especifica una configuración regional o una matriz de configuraciones regionales, y el segundo es un objeto opcional cuyas propiedades especifican exactamente qué tipo de comparación de cadenas se va a realizar. Las propiedades soportadas son las siguientes:

#### *uso*

Esta propiedad especifica cómo se va a utilizar el objeto intercalador. El valor por defecto es "ordenar", pero también se puede especificar "buscar". La idea es que, cuando se ordenan cadenas, normalmente se quiere un cotejador que diferencie tantas cadenas como sea posible para producir un ordenamiento fiable. Pero cuando se comparan dos cadenas, algunas localizaciones pueden querer una comparación menos estricta que ignore los acentos, por ejemplo.

#### *sensibilidad*

Esta propiedad especifica si el cotejador es sensible a las mayúsculas y minúsculas y a los acentos al comparar cadenas. El valor "base" provoca comparaciones que ignoran las mayúsculas y los acentos, considerando sólo la letra base de cada carácter. (Tenga en cuenta, sin embargo, que algunos idiomas consideran ciertos caracteres acentuados como letras base distintas). "acento" considera los acentos en las comparaciones pero ignora las mayúsculas y minúsculas. "case" considera las mayúsculas y minúsculas e ignora los

acentos. Y "variante" realiza comparaciones estrictas que consideran tanto las mayúsculas como los acentos. El valor por defecto de esta propiedad es "variant" cuando el uso es "sort". Si el uso es "buscar", la sensibilidad por defecto depende de la configuración regional.

#### *ignorePunctuation*

Establezca esta propiedad a true para ignorar los espacios y la puntuación al comparar cadenas. Con esta propiedad establecida a true, las cadenas "cualquiera" y "cualquiera", por ejemplo, se considerarán iguales.

#### *numérico*

Establezca esta propiedad a true si las cadenas que está comparando son enteras o contienen enteros y quiere que se ordenen en orden numérico en lugar de alfabético. Con esta opción establecida, la cadena "Versión 9" se ordenará antes que "Versión 10", por ejemplo.

#### *caseFirst*

Esta propiedad especifica qué letra debe ir primero. Si especifica "upper", entonces "A" se ordenará antes que "a". Y si especifica "minúscula", entonces "a" se ordenará antes que "A". En cualquier caso, tenga en cuenta que las variantes en mayúsculas y minúsculas de la misma letra estarán una al lado de la otra en el orden de clasificación, lo que es diferente al orden lexicográfico Unicode (el comportamiento por defecto del método Array sort()) en el que todas las letras ASCII en mayúsculas vienen antes de todas las letras ASCII en minúsculas. El valor por defecto de esta propiedad depende de la localización, y las implementaciones pueden ignorar esta propiedad y no permitirle anular el orden de las mayúsculas y minúsculas.

Una vez que haya creado un objeto Intl.Collator para la configuración regional y las opciones deseadas, puede utilizar su método compare() para comparar dos cadenas. Este método devuelve un número. Si el valor devuelto es menor que cero, entonces la primera cadena va antes que la segunda. Si es mayor que cero, entonces la primera cadena va después de la segunda. Y si compare() devuelve cero, entonces las dos cadenas son iguales en lo que respecta a este comparador.

Este método compare() que toma dos cadenas y devuelve un número menor, igual o mayor que cero es exactamente lo que el método Array sort() espera para su argumento opcional. Además, Intl.Collator vincula automáticamente el método compare() a su instancia, por lo que puede pasarlo directamente a sort() sin tener que escribir una función envolvente e invocarla a través del objeto collator. He aquí algunos ejemplos:

```
// Un comparador básico para ordenar en la configuración regional del usuario.  
// Nunca ordene cadenas legibles por humanos sin pasar algo como esto:
```

```

const collator = new Intl. Collator(). compare;
["a", "z", "A", "Z"]. sort(collator) // => ["a", "A",
"z", "Z"]

// Los nombres de los archivos suelen incluir números, por lo que debemos ordenarlos
especialmente const filenameOrder = new Intl. Collator(undefined, { numeric:
true }). compare;
["página10", "página9"]. sort(filenameOrder) // => ["página9",
"página10"]

// Encuentra todas las cadenas que coinciden con una cadena objetivo const
fuzzyMatcher = new Intl. Collator(undefined, { sensitivity: "base",
ignorePunctuation: true }). compare; let strings = ["food", "fool", "Føø Bar"];
strings.findIndex(s => fuzzyMatcher(s, "foobar") === 0) //
=> 2

```

Algunas localidades tienen más de un orden de cotejo posible. En Alemania, por ejemplo, las guías telefónicas utilizan un orden de clasificación ligeramente más fonético que el de los diccionarios. En España, antes de 1994, la "ch" y la "ll" se trataban como letras separadas, por lo que ese país tiene un orden de cotejo moderno y otro tradicional. Y en China, el orden de cotejo puede basarse en la codificación de los caracteres, en el radical base y en los trazos de cada carácter, o en la romanización Pinyin de los caracteres. Estas variantes de cotejo no pueden seleccionarse mediante el argumento de opciones de Intl.Collator, pero pueden seleccionarse añadiendo -u-co- a la cadena de configuración regional y añadiendo el nombre de la variante deseada. Utilice "de-DE-u-co-phonebk" para ordenar la guía telefónica en Alemania, por ejemplo, y "zh-TW-u-copinyin" para ordenar el Pinyin en Taiwán.

```

// Antes de 1994, CH y LL se trataban como letras separadas en
España const modernSpanish = Intl. Collator("es-ES"). compare;

const traditionalSpanish = Intl. Collator("es-ES-u-cotrad"). compare; let
palabras = ["luz", "llama", "como", "chico"]; palabras.sort(modernSpanish) //

```

```
=> ["chico", "como", "llama", "luz"] palabras.sort(traditionalSpanish) // =>  
["como", "chico", "luz", "llama"]
```

## 11.8 La API de la consola

Has visto la función `console.log()` utilizada a lo largo de este libro: en los navegadores web, imprime una cadena en la pestaña "Consola" del panel de herramientas de desarrollo del navegador, que puede ser muy útil cuando se depura. En Node, `console.log()` es una función de salida de propósito general e imprime sus argumentos al flujo `stdout` del proceso, donde normalmente aparece al usuario en una ventana de terminal como salida del programa.

La API de la consola define una serie de funciones útiles además de `console.log()`. La API no forma parte de ningún estándar de ECMAScript, pero está soportada por los navegadores y por Node y ha sido formalmente redactada y estandarizada en <https://console.spec.whatwg.org>.

La API de la consola define las siguientes funciones:

### *console.log()*

Es la más conocida de las funciones de consola. Convierte sus argumentos en cadenas de texto y los envía a la consola. Incluye espacios entre los argumentos y comienza una nueva línea después de la salida de todos los argumentos.

### *console.debug(), console.info(), console.warn(), console.error()*

Estas funciones son casi idénticas a `console.log()`. En Node, `console.error()` envía su salida al flujo `stderr` en lugar de al flujo `stdout`, pero las otras funciones son alias de

`console.log()`. En los navegadores, los mensajes de salida generados por cada una de estas funciones pueden ir precedidos de un ícono que indica su nivel o gravedad, y la consola de desarrollador también puede permitir a los desarrolladores filtrar los mensajes de la consola por nivel.

#### `console.assert()`

Si el primer argumento es verdadero (es decir, si la aserción pasa), entonces esta función no hace nada. Pero si el primer argumento es falso u otro valor falso, entonces los argumentos restantes se imprimen como si hubieran sido pasados a `console.error()` con un prefijo "Aserción fallida". Tenga en cuenta que, a diferencia de las funciones típicas de `assert()`, `console.assert()` no lanza una excepción cuando una aserción falla.

#### `console.clear()`

Esta función borra la consola cuando eso es posible. Esto funciona en los navegadores y en Node cuando Node está mostrando su salida a una terminal. Sin embargo, si la salida de Node ha sido redirigida a un archivo o a una tubería, entonces llamar a esta función no tiene ningún efecto.

#### `console.table()`

Esta función es una característica notablemente poderosa pero poco conocida para producir una salida tabular, y es particularmente útil en los programas Node que necesitan producir una salida que resuma los datos. `console.table()` intenta mostrar su argumento en forma tabular (aunque, si no puede hacerlo, lo muestra usando el formato regular de `console.log()`). Esto funciona mejor cuando el argumento es una matriz relativamente corta de objetos, y todos los objetos de la matriz tienen el mismo (relativamente pequeño) conjunto de propiedades. En este caso, cada objeto de la

matriz se formatea como una fila de la tabla, y cada propiedad es una columna de la tabla. También puede pasar una matriz de nombres de propiedades como segundo argumento opcional para especificar el conjunto de columnas deseado. Si pasa un objeto en lugar de una matriz de objetos, la salida será una tabla con una columna para los nombres de las propiedades y otra para los valores de las mismas. O, si los valores de las propiedades son a su vez objetos, sus nombres de propiedades se convertirán en columnas de la tabla.

#### `console.trace()`

Esta función registra sus argumentos como lo hace `console.log()` y, además, sigue su salida con un seguimiento de la pila. En Node, la salida va a `stderr` en lugar de `stdout`.

#### `console.count()`

Esta función toma un argumento de cadena y registra esa cadena, seguida del número de veces que ha sido llamada con esa cadena. Esto puede ser útil cuando se depura un manejador de eventos, por ejemplo, si se necesita llevar la cuenta de cuántas veces se ha disparado el manejador de eventos.

#### `console.countReset()`

Esta función toma un argumento de cadena y restablece el contador para esa cadena.

#### `console.group()`

Esta función imprime sus argumentos en la consola como si hubieran sido pasados a `console.log()`, y luego establece el estado interno de la consola para que todos los mensajes posteriores de la consola (hasta la siguiente llamada a `console.groupEnd()`) tengan una sangría relativa al mensaje que acaba de imprimir. Esto permite agrupar visualmente un grupo de mensajes relacionados con la sangría. En los

navegadores web, la consola del desarrollador suele permitir que los mensajes agrupados se contraigan y expandan como un grupo. Los argumentos de `console.group()` se utilizan normalmente para proporcionar un nombre explicativo para el grupo.

#### `console.groupCollapsed()`

Esta función funciona como `console.group()` excepto que en los navegadores web, el grupo estará "colapsado" por defecto y los mensajes que contiene estarán ocultos a menos que el usuario haga clic para expandir el grupo. En Node, esta función es un sinónimo de `console.group()`.

#### `console.groupEnd()`

Esta función no toma argumentos. No produce ninguna salida propia, pero termina la sangría y el agrupamiento causados por la llamada más reciente a `console.group()` o `console.groupCollapsed()`.

#### `console.time()`

Esta función toma un único argumento de cadena, anota la hora en que fue llamada con esa cadena y no produce ninguna salida.

#### `console.timeLog()`

Esta función toma una cadena como primer argumento. Si esa cadena se ha pasado previamente a `console.time()`, entonces imprime esa cadena seguida del tiempo transcurrido desde la llamada a `console.time()`. Si hay argumentos adicionales a `console.timeLog()`, se imprimen como si se hubieran pasado a `console.log()`.

#### `console.timeEnd()`

Esta función toma un único argumento de cadena. Si ese argumento se ha pasado previamente a `console.time()`, entonces imprime ese argumento y el tiempo transcurrido. Después de llamar a `console.timeEnd()`, ya no es legal llamar a `console.timeLog()` sin llamar primero a `console.time()` de nuevo.

### 11.8.1 Salida formateada con la consola

Las funciones de consola que imprimen sus argumentos como `console.log()` tienen una característica poco conocida: si el primer argumento es una cadena que incluye `%s`, `%i`, `%d`, `%f`, `%o`, `%O`, o `%c`, entonces este primer argumento se trata como cadena de formato,<sup>6</sup> y los valores de los argumentos posteriores se sustituyen en la cadena en lugar de las secuencias de dos caracteres `%`.

Los significados de las secuencias son los siguientes:

`%s`

El argumento se convierte en una cadena.

`%i` y `%d`

El argumento se convierte en un número y luego se trunca a un entero.

`%f`

El argumento se convierte en un número

`%o` y `%O`

El argumento se trata como un objeto, y se muestran los nombres y valores de las propiedades. (En los navegadores web, esta visualización suele ser interactiva, y los usuarios pueden expandir y contraer las propiedades para explorar una

estructura de datos anidada). Tanto %o como %O muestran los detalles del objeto. La variante en mayúsculas utiliza un formato de salida dependiente de la implementación que se considera más útil para los desarrolladores de software.

### `%c`

En los navegadores web, el argumento se interpreta como una cadena de estilos CSS y se utiliza para dar estilo a cualquier texto que siga (hasta la siguiente secuencia %c o el final de la cadena). En Node, la secuencia %c y su correspondiente argumento son simplemente ignorados.

Tenga en cuenta que a menudo no es necesario utilizar una cadena de formato con las funciones de consola: normalmente es fácil obtener una salida adecuada simplemente pasando uno o más valores (incluyendo objetos) a la función y permitiendo que la implementación los muestre de forma útil. Como ejemplo, observe que, si pasa un objeto Error a `console.log()`, se imprime automáticamente junto con su rastro de pila.

## 11.9 APIs de URLs

Dado que JavaScript es tan comúnmente utilizado en los navegadores y servidores web, es común que el código JavaScript necesite manipular las URLs. La clase URL analiza las URLs y también permite modificar (añadir parámetros de búsqueda o alterar las rutas, por ejemplo) las URLs existentes. También maneja adecuadamente el complicado tema de escapar y desescapar los distintos componentes de una URL.

La clase URL no forma parte de ningún estándar ECMAScript, pero funciona en Node y en todos los navegadores de Internet que no sean Internet Explorer. Está estandarizada en <https://url.spec.whatwg.org>.

Cree un objeto URL con el constructor URL(), pasando una cadena de URL absoluta como argumento. O pasar una URL relativa como primer argumento y la URL absoluta a la que es relativa como segundo argumento. Una vez creado el objeto URL, sus diversas propiedades le permiten consultar las versiones sin esconder de las distintas partes de la URL:

```
let url = new URL("https://example.com:8000/path/name?  
q=term#fragment"); url.href //=> "https://example.com:8000/path/name?  
q=term#fragment" url.origin //=> "https://example.com:8000" url.  
protocol //=> "https:" url.host //=> "ejemplo.com:8000" url.  
hostname //=> "ejemplo.com" url.port //=> "8000" url.  
pathname //=> "/ruta/nombre" url.search //=> "?q=term" url.hash  
//=> "#fragment"
```

Aunque no es de uso común, las URLs pueden incluir un nombre de usuario o un nombre de usuario y una contraseña, y la clase URL puede analizar también estos componentes de la URL:

```
let url = new URL("ftp://admin:1337!@ftp.example.com/"); url.href //=>  
"ftp://admin:1337!@ftp.example.com/" url.origin //=>  
"ftp://ftp.example.com" url.username //=> "admin" url.password //=>  
"1337!"
```

La propiedad de origen aquí es una simple combinación del protocolo de la URL y el host (incluyendo el puerto si se especifica uno). Como tal, es una propiedad de sólo lectura. Pero cada una de

las otras propiedades demostradas en el ejemplo anterior es de lectura/escritura: puede establecer cualquiera de estas propiedades para establecer la parte correspondiente de la URL:

```
let url = new URL("https://example.com"); // Empezar con la url de nuestro servidor.  
 pathname = "api/search"; // Añadir una ruta a una url de punto final de la API.  
 search = "q=test"; // Añadir un parámetro de consulta  
 url.toString() // => "https://example.com/api/search?q=test"
```

Una de las características importantes de la clase URL es que añade correctamente la puntuación y escapa de los caracteres especiales en las URL cuando es necesario:

```
let url = new URL("https://example.com"); url.pathname =  
 "path with spaces"; url.search = "q=foo#bar"; url.pathname  
 // => "/path%20with%20spaces" url.search // =>  
 "?q=foo%23bar" url.href // =>  
 "https://example.com/path%20with%20spaces?q=foo%23bar"
```

La propiedad href en estos ejemplos es especial: leer href es equivalente a llamar a `toString()`: reensambla todas las partes de la URL en la forma de cadena canónica de la URL. Y al establecer href a una nueva cadena se vuelve a ejecutar el analizador de URL en la nueva cadena como si se hubiera llamado de nuevo al constructor de `URL()`.

En los ejemplos anteriores, hemos estado utilizando la propiedad de búsqueda para referirnos a toda la parte de la consulta de una URL, que consiste en los caracteres que van desde un signo de interrogación hasta el final de la URL o hasta el primer carácter de hash. A veces, es suficiente con tratar esto como una sola propiedad de la URL. Sin embargo, a menudo las solicitudes HTTP codifican los valores de varios campos de formulario o varios

parámetros de la API en la parte de consulta de una URL utilizando la propiedad application/x-www-form-urlencoded formato. En este formato, la parte de la consulta de la URL es un signo de interrogación seguido de uno o más pares de nombre/valor, que están separados entre sí por ampersands. El mismo nombre puede aparecer más de una vez, dando lugar a un parámetro de búsqueda con nombre con más de un valor.

Si quieras codificar este tipo de pares nombre/valor en la parte de la consulta de una URL, entonces la propiedad searchParams será más útil que la propiedad search. La propiedad search es una cadena de lectura/escritura que permite obtener y establecer toda la porción de consulta del URL. La propiedad searchParams es una referencia de sólo lectura a un objeto URLSearchParams, que tiene una API para obtener, establecer, añadir, eliminar y ordenar los parámetros codificados en la parte de consulta de la URL:

```
let url = new URL("https://example.com/search"); url.search //=> "": aún no hay  
consulta url.SearchParams.append("q", "term"); // Añadir un parámetro de  
búsqueda url.search //=> "?q=termino" url.SearchParams.set("q", "x"); //  
Cambiar el valor de este parámetro url.search //=> "?q=x" url.SearchParams.  
get("q") //=> "x": consultar el valor del parámetro url.SearchParams.has("q") //  
=> true: hay un parámetro q url.SearchParams.has("p") //=> false: no hay  
parámetro p url.SearchParams.append("opts", "1"); // Añadir otro parámetro de  
búsqueda url.search //=> "?q=x&opts=1" url.SearchParams.append("opts", "&");  
// Añadir otro valor para el mismo nombre url.search //=> "?  
q=x&opts=1&opts=%26": nota escape url.SearchParams.get("opts") //=> "1": el  
primer valor
```

```
urlSearchParams.getAll("opts") //=> ["1", "&"]: todos los valores url.searchParams.sort(); // Poner los parámetros en orden alfabético url.search //=> "?opts=1&opts=%26&q=x" urlSearchParams.set("opts", "y"); // Cambiar el parámetro opts url.search //=> "?opts=y&q=x" // searchParams es iterable [... urlSearchParams] //=> [{"opts": "y"}, {"q": "x"}] urlSearchParams.delete("opts"); // Eliminar el parámetro opts url.search //=> "?q=x" url.href //=> "https://example.com/search?q=x"
```

El valor de la propiedad `searchParams` es un objeto `URLSearchParams`. Si quieres codificar los parámetros de la URL en una cadena de consulta, puedes crear un objeto `URLSearchParams`, añadir los parámetros, luego convertirlo en una cadena y establecerlo en la propiedad de búsqueda de una URL:

```
let url = new URL("http://example.com"); let params = new URLSearchParams(); params.append("q", "term"); params.append("opts", "exact"); params.toString() //=> "q=term&opts=exact" url.search = params; url.href //=> "http://example.com/? q=term&opts=exact"
```

### 11.9.1 Funciones de URL heredadas

Antes de la definición de la API de URL descrita anteriormente, ha habido múltiples intentos de soportar el escape y el desescapado de URL en el núcleo del lenguaje JavaScript. El primer intento fueron las funciones definidas globalmente `escape()` y `unescape()`, que ahora están obsoletas pero que siguen siendo ampliamente implementadas. No deberían utilizarse.

Cuando `escape()` y `unescape()` quedaron obsoletos, ECMAScript introdujo dos pares de funciones globales alternativas:

### *`encodeURI()` y `decodeURI()`*

`encodeURI()` toma una cadena como argumento y devuelve una nueva cadena en la que se escapan los caracteres no ASCII y ciertos caracteres ASCII (como el espacio). `decodeURI()` invierte el proceso.

Los caracteres que deben escaparse se convierten primero a su codificación UTF8, y luego cada byte de esa codificación se sustituye por una secuencia de escape %xx, donde xx son dos dígitos hexadecimales. Dado que `encodeURI()` está pensada para codificar URLs enteras, no escapa a los caracteres separadores de URL como /, ? y #. Pero esto significa que `encodeURI()` no puede funcionar correctamente para las URL que tienen esos caracteres dentro de sus diversos componentes.

### *`encodeURIComponent()` y `decodeURIComponent()`*

Este par de funciones funciona igual que `encodeURI()` y `decodeURI()`, salvo que están pensadas para escapar de los componentes individuales de un URI, por lo que también escapan de caracteres como /, ? y # que se utilizan para separar esos componentes. Estas son las más útiles de las funciones de URL heredadas, pero tenga en cuenta que `encodeURIComponent()` escapará caracteres / en un nombre de ruta que probablemente no quiera escapar. Y convertirá los espacios en un parámetro de consulta a %20, aunque se supone que los espacios deben ser escapados con un + en esa parte de una URL.

El problema fundamental de todas estas funciones heredadas es que pretenden aplicar un único esquema de codificación a todas las partes de una URL cuando el hecho es que las diferentes partes de una URL utilizan codificaciones diferentes. Si quieres una URL correctamente formateada y codificada, la solución es simplemente usar la clase URL para toda la manipulación de la URL que hagas.

## 11.10 Temporizadores

Desde los primeros días de JavaScript, los navegadores web han definido dos funciones-setTimeout() y setInterval()-que permiten para pedir al navegador que invoque una función una vez transcurrido un tiempo determinado o que invoque la función repetidamente en un intervalo especificado. Estas funciones nunca se han estandarizado como parte del núcleo del lenguaje, pero funcionan en todos los navegadores y en Node y son una parte de facto de la biblioteca estándar de JavaScript.

El primer argumento de setTimeout() es una función, y el segundo argumento es un número que especifica cuántos milisegundos deben transcurrir antes de que la función sea invocada. Después de la cantidad de tiempo especificada (y quizás un poco más si el sistema está ocupado), la función será invocada sin argumentos. Aquí, por ejemplo, hay tres llamadas a setTimeout() que imprimen mensajes de consola después de un segundo, dos segundos y tres segundos:

```
setTimeout(() => { console.log("Listo..."); }, 1000); setTimeout(() => { console.log("listo..."); }, 2000); setTimeout(() => { console.log("iya!"); }, 3000);
```

Observe que `setTimeout()` no espera a que transcurra el tiempo antes de regresar. Las tres líneas de código de este ejemplo se ejecutan casi instantáneamente, pero luego no ocurre nada hasta que transcurren 1.000 milisegundos.

Si se omite el segundo argumento de `setTimeout()`, se pone por defecto a 0. Eso no significa, sin embargo, que la función que se especifica sea invocada inmediatamente. En cambio, la función se registra para ser llamada "tan pronto como sea posible". Si un navegador está particularmente ocupado manejando la entrada del usuario u otros eventos, puede tomar 10 milisegundos o más antes de que la función sea invocada.

`setTimeout()` registra una función para ser invocada una vez. A veces, esa función llamará a su vez a `setTimeout()` para programar otra invocación en un momento futuro. Sin embargo, si quiere invocar una función repetidamente, suele ser más sencillo utilizar `setInterval()`.

`setInterval()` toma los mismos dos argumentos que `setTimeout()` pero invoca la función repetidamente cada vez que el número especificado de milisegundos (aproximadamente) ha transcurrido.

Tanto `setTimeout()` como `setInterval()` devuelven un valor. Si usted guardar este valor en una variable, puede usarlo después para cancelar la ejecución de la función pasándolo a `clearTimeout()` o `clearInterval()`. El valor devuelto es típicamente un número en los navegadores web y es un objeto en Node. El tipo real no importa, y debes tratarlo como un valor opaco. Lo único que puede hacer con este valor es pasarlo a `clearTimeout()` para cancelar la ejecución de una función registrada con `setTimeout()` (suponiendo que no haya

sido invocada todavía) o para detener la ejecución repetida de una función registrada con `setInterval()`.

Aquí hay un ejemplo que demuestra el uso de `setTimeout()`, `setInterval()`, y `clearInterval()` para mostrar un simple reloj digital con la API de la consola:

```
// Una vez por segundo: borra la consola e imprime la hora actual let clock =  
setInterval(() => { console.clear();  
  console.log(new Date().toLocaleTimeString()); }, 1000);  
  
// Después de 10 segundos: detener el código repetitivo anterior. setTimeout(() => {  
  clearInterval(clock); }, 10000);
```

Veremos `setTimeout()` y `setInterval()` de nuevo cuando cubramos la programación asíncrona en el [capítulo 13](#).

## 11.11 Resumen

Aprender un lenguaje de programación no consiste sólo en dominar la gramática. Es igualmente importante estudiar la biblioteca estándar para familiarizarse con todas las herramientas que vienen con el lenguaje. Este capítulo ha documentado la biblioteca estándar de JavaScript, que incluye:

- Estructuras de datos importantes, como Set, Map y arrays
- tipados.
- Las clases Date y URL para trabajar con fechas y URLs.

La gramática de expresiones regulares de JavaScript y su clase `RegExp` para la coincidencia de patrones textuales.

- 
- Biblioteca de internacionalización de JavaScript para formatear fechas, horas y números y para ordenar cadenas.
  - El objeto JSON para serializar y deserializar estructuras de datos simples y el objeto consola para registrar mensajes.

No todo lo documentado aquí está definido por la especificación del lenguaje JavaScript:

<sup>1</sup> algunas de las clases y funciones documentadas aquí fueron implementadas primero en los navegadores web y luego adoptadas por Node, convirtiéndolas en miembros de facto de la biblioteca estándar de JavaScript.

Este orden de iteración predecible es otra cosa de los conjuntos de JavaScript que los programadores de Python <sup>2</sup> pueden encontrar sorprendente.

Las matrices tipificadas se introdujeron por primera vez en JavaScript del lado del cliente cuando los navegadores web añadieron

<sup>3</sup> soporte para gráficos WebGL. La novedad en ES6 es que se han elevado a una característica central del lenguaje.

Excepto dentro de una clase de caracteres (corchetes), donde \b coincide con el carácter de retroceso <sup>4</sup>.

Analizar las URLs con expresiones regulares no es una buena idea. Véase §11.9 para un analizador de URLs más robusto:

Los programadores de C reconocerán muchas de estas secuencias de caracteres de la función printf() <sup>6</sup>.



# Capítulo 12. Iteradores y generadores

---

Los objetos iterables y sus iteradores asociados son una característica de ES6 que hemos visto varias veces a lo largo de este libro. Los arrays (incluyendo TypedArrays) son iterables, al igual que las cadenas y los objetos Set y Map. Esto significa que el contenido de estas estructuras de datos puede ser iterado - repetido- con el bucle for/of, como vimos en §5.4.4:

```
let sum = 0; for(let i of [1,2,3]) { // Hacer un bucle por cada uno de estos valores sum +=  
    i;  
} suma // => 6
```

Los iteradores también pueden utilizarse con el operador ... para expandir o "extender" un objeto iterable en un inicializador de matriz o en una invocación de función, como vimos en §7.1.2:

```
let chars = [... "abcd"]; // chars == ["a", "b", "c", "d"] let data = [1, 2, 3, 4, 5];  
Math.max(... data) // => 5
```

Los iteradores se pueden utilizar con la asignación de desestructuración:

```
let purpleHaze = Uint8Array.of(255, 0, 255, 128); let [r, g, b, a] =  
purpleHaze; // a == 128
```

Cuando se itera un objeto Map, los valores devueltos son pares [clave, valor], que funcionan bien con la asignación de desestructuración en un bucle for/of:

```
let m = new Map([["uno", 1], ["dos", 2]]); for(let [k,v] of m) console.log(k, v); //  
Registra 'uno 1' y 'dos 2'
```

Si quieres iterar sólo las claves o sólo los valores en lugar de los pares, puedes utilizar los métodos keys() y values():

*[... m] //=> [[“uno”, 1], [“dos”, 2]]: iteración por defecto [... m. entries()] //=> [[“uno”, 1], [“dos”, 2]]: el método entries() es el mismo ..... m. keys()] //=> [“uno”, “dos”]: el método keys() sólo itera el mapa de claves [... m. values()] //=> [1, 2]: el método values() sólo itera el mapa de valores*

Por último, una serie de funciones y constructores incorporados que se utilizan habitualmente con los objetos Array están escritos (en ES6 y posteriores) para aceptar iteradores arbitrarios en su lugar. El constructor Set() es una de estas API:

```
// Las cadenas son iterables, por lo que los dos conjuntos son iguales: new  
Set("abc") //=> new Set(["a", "b", "c"])
```

Este capítulo explica cómo funcionan los iteradores y demuestra cómo crear tus propias estructuras de datos que sean iterables. Después de explicar los iteradores básicos, este capítulo cubre los generadores, una nueva y poderosa característica de ES6 que se utiliza principalmente como una forma particularmente fácil de crear iteradores.

## 12.1 Cómo funcionan los iteradores

El bucle for/of y el operador de propagación funcionan a la perfección con los objetos iterables, pero vale la pena entender lo

que realmente sucede para que la iteración funcione. Hay tres tipos distintos que debes entender para comprender la iteración en JavaScript. Primero, están los objetos *iterables*: son tipos como Array, Set y Map que pueden ser iterados. En segundo lugar, está el objeto *iterador* en sí, que realiza la iteración. Y tercero, está el objeto *resultado de la iteración* que contiene el resultado de cada paso de la iteración.

Un objeto *iterable* es cualquier objeto con un método iterador especial que devuelve un objeto iterador. Un *iterador* es cualquier objeto con un método `next()` que devuelve un objeto resultado de iteración. Y un objeto resultado de *iteración* es un objeto con propiedades denominadas `value` y `done`. Para iterar un objeto iterable, primero se llama a su método iterador para obtener un objeto iterador. Luego, llamas al método `next()` del objeto iterador repetidamente hasta que el valor devuelto tenga su propiedad `done` establecida a `true`. Lo complicado de esto es que el método iterador de un objeto iterable no tiene un nombre convencional sino que utiliza el símbolo `Symbol.iterator` como nombre. Así que un simple bucle `for/of` sobre un objeto iterable también podría escribirse de la manera más difícil, así

```
let iterable = [99]; let iterator = iterable[Symbol.iterator](); for(let result = iterator.next(); !result.done; result = iterator.next()) { console.log(result.value) // result.value == 99 }
```

El objeto iterador de los tipos de datos iterables incorporados es en sí mismo iterable. (Es decir, tiene un método llamado `Symbol.iterator` que sólo se devuelve a sí mismo.) Esto es ocasionalmente útil en código como el siguiente cuando se quiere iterar a través de un iterador "parcialmente utilizado":

```
let list = [1,2,3,4,5]; let iter = list[Symbol.iterator](); let head = iter.next().value; // head == 1 let tail = [... iter]; // tail == [2,3,4,5]
```

## 12.2 Implementación de objetos iterables

Los objetos iterables son tan útiles en ES6 que deberías considerar hacer iterables tus propios tipos de datos siempre que representen algo que pueda ser iterado. Las clases Range mostradas en los Ejemplos 9-2 y 9-3 del Capítulo 9 eran iterables. Esas clases usaron funciones generadoras para hacerse iterables. Documentaremos los generadores más adelante en este capítulo, pero primero, implementaremos la clase Range una vez más, haciéndola iterable sin depender de un generador.

Para que una clase sea iterable, debe implementar un método cuyo nombre sea el símbolo `Symbol.iterator`. Ese método debe devolver un objeto iterador que tenga un método `next()`. Y el método `next()` debe devolver un objeto resultado de la iteración que tenga una propiedad `value` y/o una propiedad boolean `done`. El Ejemplo 12-1 implementa una clase iterable `Range` y demuestra cómo crear objetos iterables, iteradores y resultados de iteración.

### Ejemplo 12-1. Una clase Range numérica iterable

---

```
/*
 * Un objeto Range representa un rango de números {x: desde <= x
 <= a}
```

*Range define un método has() para comprobar si un número dado es un miembro \* del rango. Range es iterable e itera todos los enteros dentro del rango. \*/*

```
class Range {
    constructor (from, to) { this.from = from; this.to = to; }

    // Hacer que un Rango actúe como un Conjunto de números has(x) { return typeof x === "number" && this.from <= x && x <= this.to; }

    // Devuelve la representación en forma de cadena del rango utilizando la notación de conjunto toString() { devuelve `{ x | ${this.from} ≤ x ≤ ${this.to} }` };

    // Hacer un Range iterable devolviendo un objeto iterador.
    // Observe que el nombre de este método es un símbolo especial, no una cadena.
    [Símbolo. iterador]() {
        // Cada instancia del iterador debe iterar el rango independientemente de
        // otros. Así que necesitamos una variable de estado para seguir nuestra ubicación
        // en la // iteración. Empezamos en el primer entero >= desde. let next = Math. ceil(this.
        from); // Este es el siguiente valor que devolvemos let last = this. to; // No devolveremos
        // nada > this return { // Este es el objeto iterador // Este método next() es lo que hace que
        // sea un objeto iterador. // Debe devolver un objeto iterador resultado. next() {
        return (next <= last) // Si aún no hemos devuelto el último valor ? { valor: next++ } //
        // devuelve el siguiente valor y lo incrementa
        : { done: true }; // en caso contrario indicar
    }
}
```

```

que hemos terminado.
},
// Por comodidad, hacemos que el propio iterador sea iterable. [Símbolo.
iterador](){ devuelve esto; }
};

}

for(let x of new Range(1,10)) console.log(x); // Registra los números del 1 al 10 [... new
Range(-2,2)] //=> [-2, -1, 0,
1, 2]

```

Además de hacer sus clases iterables, puede ser muy útil definir funciones que devuelvan valores iterables. Considere estas alternativas basadas en iterables a los métodos map() y filter() de las matrices de JavaScript:

```

// Devuelve un objeto iterable que itera el resultado de aplicar f() // a cada valor del
iterable fuente function map(iterable, f) { let iterator = iterable[Symbol.iterator](); return {
// Este objeto es tanto iterador como iterable [Symbol.iterator]() { return this; }, next() {
let v = iterator.next(); if (v.done) { return v; } else { return { value: f(v.value) }; }
}
};
}

// Mapear un rango de enteros a sus cuadrados y convertirlo en un array
[... map(new Range(1,4), x => x*x)] //=> [1, 4, 9, 16]

// Devuelve un objeto iterable que filtra el iterable especificado,
// iterar sólo los elementos para los que el predicado devuelve verdadero function
filter(iterable, predicate) { let iterator = iterable[Symbol.iterator](); return { // Este
objeto es tanto iterador como iterable [Symbol.iterator]() { return this; }, next() { for(;;) {
let v = iterator.next(); if (v.done | predicate(v.value)) { return v; }
}
}
};
}

// Filtrar un rango para que nos quedemos sólo con los números pares
[... filter(new Range(1,10), x => x % 2 === 0)] //=>
[2,4,6,8,10]

```

Una característica clave de los objetos iterables y de los iteradores es que son intrínsecamente perezosos: cuando se requiere un cálculo para calcular el siguiente valor, ese cálculo puede ser diferido hasta que el valor sea realmente necesario. Supongamos, por ejemplo, que tiene una cadena de texto muy larga que quiere dividir en palabras separadas por espacios. Podría simplemente utilizar el método `split()` de su cadena, pero si hace esto, entonces toda la cadena tiene que ser procesada antes de que pueda utilizar incluso la primera palabra. Y terminas asignando mucha memoria para el array devuelto y todas las cadenas dentro de él. Aquí hay una función que le permite iterar perezosamente las palabras de una cadena sin mantenerlas todas en memoria a la vez (en ES2020, esta función sería mucho más fácil de implementar usando el método `matchAll()` que devuelve el iterador descrito en [§11.3.2](#)):

```

function words(s) { var r = /\s+|$/g; // Coincidir con uno o más espacios o final
  r.lastIndex = s.match(/\^/).index; // Empieza a coincidir en el primer no-espacio
  return { // Devuelve un objeto iterable [Symbol.iterator]()
    let start = r.lastIndex; // Reanuda donde terminó la última coincidencia
    if (start < s.length) { // Si no hemos terminado
      let match = r.exec(s); // Coincide con el siguiente límite de la palabra
      if (match) { // Si encontramos una, devuelve la palabra
        return { value: s.substring(start, match.index) };
      }
    }
  }
}

[... words(" abc def ghi! ")] //=> ["abc", "def", "ghi!"]

```

### 12.2.1 "Cerrar" un Iterador: El método de retorno

Imagine una variante (del lado del servidor) de JavaScript del iterador words()

que, en lugar de tomar una cadena de origen como argumento, toma el nombre de un archivo, abre el archivo, lee líneas de él, e itera las palabras de esas líneas. En la mayoría de los sistemas operativos, los programas que abren archivos para leer de ellos necesitan recordar cerrar esos archivos cuando terminan de leer,

por lo que este hipotético iterador se aseguraría de cerrar el archivo después de que el método next() devuelva la última palabra en él.

Pero los iteradores no siempre se ejecutan hasta el final: un bucle for/of puede terminar con un break o return o por una excepción. Del mismo modo, cuando se utiliza un iterador con asignación de desestructuración, el método next() sólo se llama las veces suficientes para obtener valores para cada una de las variables especificadas. El iterador puede tener muchos más valores que podría devolver, pero nunca serán solicitados.

Si nuestro hipotético iterador palabras-en-un-archivo nunca llega hasta el final, todavía necesita cerrar el archivo que abrió. Por esta razón, los objetos iteradores pueden implementar un método return() para acompañar al método next(). Si la iteración se detiene antes de que next() haya devuelto un resultado de iteración con la propiedad done puesta a true (más comúnmente porque usted salió de un bucle for/of antes de tiempo mediante una sentencia break), entonces el intérprete comprobará si el objeto iterador tiene un método return(). Si este método existe, el intérprete lo invocará sin argumentos, dándole al iterador la oportunidad de cerrar archivos, liberar memoria, y limpiar de alguna manera después de sí mismo. El método return() debe devolver un objeto resultado del iterador. Las propiedades del objeto se ignoran, pero es un error devolver un valor que no sea un objeto.

El bucle for/of y el operador de propagación son características realmente útiles de JavaScript, por lo que cuando se crean APIs, es una

buenas ideas utilizarlos siempre que sea posible. Pero tener que trabajar con un objeto iterable, su objeto iterador y los objetos resultado del iterador hace el proceso algo complicado. Afortunadamente, los generadores pueden simplificar drásticamente la creación de iteradores personalizados, como veremos en el resto de este capítulo.

## 12.3 Generadores

Un *generador* es un tipo de iterador definido con la nueva y potente sintaxis de ES6; es especialmente útil cuando los valores a iterar no son los elementos de una estructura de datos, sino el resultado de un cálculo.

Para crear un generador, primero hay que definir una *función generadora*. Una función generadora es sintácticamente como una función normal de JavaScript, pero se define con la palabra clave `function*` en lugar de `function`. (Técnicamente, no se trata de una nueva palabra clave, sino de un \* después de la palabra clave `function` y antes del nombre de la función). Cuando se invoca una función generadora, ésta no ejecuta realmente el cuerpo de la función, sino que devuelve un objeto generador. Este objeto generador es un iterador. Llamar a su método `next()` hace que el cuerpo de la función generadora se ejecute desde el principio (o cualquiera que sea su posición actual) hasta que llegue a una sentencia `yield`. `yield` es nuevo en ES6 y es algo así como una sentencia `return`. El valor de la sentencia `yield` se convierte en el valor devuelto por la llamada `next()` en el iterador. Un ejemplo aclara esto:

```
// Una función generadora que produce el conjunto de un dígito
```

```

(base-10) primos. function* oneDigitPrimes() { // La invocación de esta función no
ejecuta el código

    yield 2; // pero sólo devuelve un objeto generador. Al llamar a yield 3; // el método
    next() de ese generador se ejecuta yield 5; // el código hasta que una sentencia yield
    proporciona yield 7; // el valor de retorno del método next(). }

// Cuando invocamos la función generadora, obtenemos un generador let primes =
oneDigitPrimes();

// Un generador es un objeto iterador que itera los valores producidos primos.
next(). valor // => 2 primos. next(). valor // => 3 primos. next(). valor // => 5
primos. next(). valor // => 7 primos. next(). hecho // => verdadero

// Los generadores tienen un método Symbol.iterator para hacerlos iterables
primes[Symbol.iterator]() // => primes

// Podemos usar generadores como otros tipos iterables [...]
oneDigitPrimes() // => [2,3,5,7] let sum = 0; for(let prime of
oneDigitPrimes()) sum += prime; sum // => 17

```

En este ejemplo, hemos utilizado una sentencia **function\*** para definir un generador. Sin embargo, al igual que las funciones regulares, también podemos definir generadores en forma de expresión. Una vez más, simplemente ponemos un asterisco después de la palabra clave **function**:

```
const seq = function*(from,to) { for(let i = from; i <= to; i++) yield i; };
[... seq(3,5)] // => [3, 4, 5]
```

En las clases y en los literales de los objetos, podemos utilizar la notación abreviada para omitir completamente la palabra clave

function cuando definimos métodos. Para definir un generador en este contexto, simplemente utilizamos un asterisco antes del nombre del método donde habría estado la palabra clave de función, si la hubiéramos utilizado:

```
dejemos o = { x: 1, y: 2, z: 3,
    // Un generador que produce cada una de las claves de este objeto *g() {
    for(let key of Object.keys(this)) { yield key; }
}
[... o. g()] //=> ["x", "y", "z", "g"]
```

Tenga en cuenta que no hay manera de escribir una función generadora utilizando la sintaxis de la función de flecha.

Los generadores suelen facilitar la definición de clases iterables.

Podemos sustituir el método [Symbol.iterator]() que aparece en

Ejemplo 12-1 con un \* mucho más corto

[Symbol.iterator&rbrack;() función generadora que tiene este aspecto:

```
*[Symbol.iterator]() { for(let x = Math.ceil(this.from); x <= this.to; x++) yield
x; }
```

Vea Ejemplo 9-3 en el Capítulo 9 para ver esta función iteradora basada en el generador en su contexto.

### 12.3.1 Ejemplos de generadores

Los generadores son más interesantes si realmente *generan* los valores que producen haciendo algún tipo de cálculo. Aquí, por

ejemplo, hay una función generadora que produce números Fibonacci:

```
function* fibonacciSequence() { let x = 0, y = 1; for(;;) { yield y; [x, y] = [y, x+y]; // Nota: asignación desestructurada }
```

Obsérvese que la función generadora fibonacciSequence() tiene aquí un bucle infinito y arroja valores para siempre sin regresar. Si este generador se utiliza con el operador de propagación ..., hará un bucle hasta que se agote la memoria y el programa se bloquee. Sin embargo, con cuidado, es posible utilizarlo en un bucle for/of:

```
// Devuelve el enésimo número de Fibonacci function fibonacci(n) {
for(let f of fibonacciSequence()) { if (n-- <= 0) return f; } }
fibonacci(20) // => 10946
```

Este tipo de generador infinito se vuelve más útil con un generador take() como este:

```
// Obtener los primeros n elementos del objeto iterable especificado function* take(n, iterable) { let it = iterable[Symbol.iterator](); // Obtener el iterador para objeto iterable while(n-- > 0) { // Bucle n veces:
let next = it.next(); // Obtener el siguiente elemento del iterador.
if (next.done) return; // Si no hay más valores, retornar antes de tiempo else yield next.value; // en caso contrario, ceder el valor }
}

// Un array de los 5 primeros números de Fibonacci
[... take(5, fibonacciSequence())] // => [1, 1, 2, 3, 5]
```

Aquí hay otra función generadora útil que intercala los elementos de múltiples objetos iterables:

```
// Dada una matriz de iterables, cede sus elementos en orden intercalado.
```

```

function* zip(... iterables) { // Obtener un iterador para cada iterable
  let iterables = iterables.map(i => i[Symbol.iterator]());
  let index = 0;
  while(iterables.length > 0) {
    Mientras haya algunos iteradores if (index >= iterables.length) { // Si llegamos al
    último iterador index = 0; // volver al primero. } let item = iterables[index].next();
    // Obtener el siguiente elemento del siguiente iterador. if (item.done) { // Si ese
    iterador está terminado iterables.splice(index, 1); // entonces eliminarlo del array.
  } else { // En caso contrario, cede item.value; // cede el valor iterado index++; // y pasa
    al siguiente iterador.
  }
}
}

// Intercalar tres objetos iterables
[... zip(oneDigitPrimes(), "ab", [0])] // =>
[2, "a", 0, 3, "b", 5, 7]

```

### 12.3.2 Generadores de rendimiento\* y recursivos

Además del generador zip() definido en el ejemplo anterior, podría ser útil disponer de una función generadora similar que arroje los elementos de múltiples objetos iterables de forma secuencial en lugar de intercalarlos. Podríamos escribir ese generador así

```

function* sequence(... iterables) {
  for(let iterable of iterables) {
    for(let item of iterable) {
      yield item;
    }
  }
}

[... secuencia("abc", oneDigitPrimes())] // =>
["a", "b", "c", 2, 3, 5, 7]

```

Este proceso de ceder los elementos de algún otro objeto iterable es lo suficientemente común en las funciones generadoras como para que ES6 tenga una sintaxis especial para ello. La palabra clave yield\* es como yield, excepto que, en lugar de devolver un solo valor, itera un objeto iterable y devuelve cada uno de los valores

resultantes. La función generadora sequence() que hemos utilizado puede simplificarse con yield\* de la siguiente manera:

```
function* sequence(... iterables) { for(let iterable of  
iterables) {  
  
    yield* iterable;  
}  
  
[... secuencia("abc",oneDigitPrimes())] // =>  
["a", "b", "c",2,3,5,7]
```

El método forEach() de la matriz es a menudo una forma elegante de hacer un bucle sobre los elementos de una matriz, por lo que podría estar tentado de escribir la función sequence() de esta manera:

```
function* sequence(... iterables) {  
    iterables.forEach(iterable => yield* iterable ); // Error  
}
```

Sin embargo, esto no funciona. yield y yield\* sólo pueden utilizarse dentro de funciones generadoras, pero la función de flecha anidada en este código es una función regular, no una función generadora de funciones\*, por lo que yield no está permitido.

yield\* puede utilizarse con cualquier tipo de objeto iterable, incluyendo los iterables implementados con generadores. Esto significa que yield\* nos permite definir generadores recursivos, y podrías utilizar esta característica para permitir una simple iteración no recursiva sobre una estructura de árbol definida recursivamente, por ejemplo.

## 12.4 Funciones avanzadas del generador

El uso más común de las funciones generadoras es crear iteradores, pero la característica fundamental de los generadores es que nos permiten pausar un cálculo, obtener resultados intermedios y reanudar el cálculo más tarde. Esto significa que los generadores tienen características que van más allá de las de los iteradores, y exploramos esas características en las siguientes secciones.

### 12.4.1 El valor de retorno de una función generadora

Las funciones generadoras que hemos visto hasta ahora no han tenido declaraciones de retorno, o si las han tenido, se han utilizado para provocar un retorno anticipado, no para devolver un valor. Sin embargo, como cualquier función, una función generadora puede devolver un valor. Para entender lo que ocurre en este caso, recuerda cómo funciona la iteración. El valor de retorno de la función `next()` es un objeto que tiene una propiedad `value` y/o una propiedad `done`. Con los iteradores y generadores típicos, si la propiedad `value` está definida, entonces la propiedad `done` está indefinida o es falsa. Y si `done` es verdadero, entonces `value` es indefinido. Pero en el caso de un generador que devuelve un valor, la llamada final a `next` devuelve un objeto que tiene definidos tanto `value` como `done`. La propiedad `value` contiene el valor de retorno de la función generadora, y la propiedad `done` es `true`, indicando que no hay más valores para iterar. Este valor final es ignorado por el bucle `for/of` y por el operador `spread`, pero está disponible para el código que itera manualmente con llamadas explícitas a `next()`:

```

function *oneAndDone() { yield 1;
return "done"; }

// El valor de retorno no aparece en la iteración normal.
[... oneAndDone()] // => [1]

// Pero está disponible si llamas explícitamente a next() let generator =
oneAndDone(); generator. next() // => { valor: 1, hecho: falso} generator. next() // =>
{ valor: "hecho", hecho: verdadero
}
// Si el generador ya está hecho, el valor de retorno no se devuelve de nuevo
generator. next() // => { value: undefined, done: true }

```

## 12.4.2 El valor de una expresión de rendimiento

En la discusión anterior, hemos tratado a `yield` como una declaración que toma un valor pero que no tiene valor propio. Sin embargo, `yield` es una expresión que puede tener un valor.

Cuando se invoca el método `next()` de un generador, la función del generador se ejecuta hasta llegar a una expresión `yield`. La expresión que sigue a la palabra clave `yield` se evalúa, y ese valor se convierte en el valor de retorno de la invocación a `next()`. En este punto, la función generadora deja de ejecutarse justo en medio de la evaluación de la expresión `yield`. La próxima vez que se llame al método `next()` del generador, el argumento pasado a `next()` se convierte en el valor de la expresión `yield` que estaba en pausa. Así que el generador devuelve valores a su llamador con `yield`, y el llamador pasa valores al generador con `next()`. El generador y el llamador son dos flujos de ejecución separados que pasan valores (y control) de un lado a otro. El siguiente código lo ilustra:

```

function* smallNumbers() {
  console. log("next() invocado la primera vez; argumento descartado"); let y1 =
yield 1; // y1 == "b"

  console. log("next() invocado por segunda vez con argumento", y1); let y2 = yield 2; //
y2 == "c"

  console. log("next() invocado por tercera vez con argumento", y2); let y3 = yield 3; //
y3 == "d"

  console. log("next() invocado por cuarta vez con argumento", y3); return 4; }

let g = smallNumbers(); console. log("generador creado; aún no se ha ejecutado
ningún código"); let n1 = g. next("a"); // n1.value == 1 console. log("generador
rendido", n1.value); let n2 = g. next("b"); // n2.value == 2 console. log("generator
yielded", n2.value); let n3 = g. next("c"); // n3.value == 3 console. log("generator
yielded", n3.value); let n4 = g. next("d"); // n4 == { value: 4, hecho: verdadero }
console. log("generador rendido", n4.valor);

```

Cuando este código se ejecuta, produce la siguiente salida que demuestra el vaivén entre los dos bloques de código:

generador creado; aún no se ejecuta el código next() invocado la primera vez; argumento descartado generador devuelto 1 next() invocado una segunda vez con el argumento b generador devuelto 2 next() invocado una tercera vez con el argumento c generador devuelto 3 next() invocado una cuarta vez con el argumento d generador devuelto 4

Nótese la asimetría de este código. La primera invocación de next() inicia el generador, pero el valor pasado a esa invocación no es accesible para el generador.

### 12.4.3 Los métodos return() y throw() de un generador

Hemos visto que puedes recibir valores producidos o devueltos por una función generadora. Y puedes pasar valores a un generador en ejecución pasando esos valores cuando llamas al método `next()` del generador.

Además de proporcionar entrada a un generador con `next()`, también puede alterar el flujo de control dentro del generador llamando a sus métodos `return()` y `throw()`. Como sus nombres sugieren, llamar a estos métodos en un generador hace que éste devuelva un valor o lance una excepción como si la siguiente sentencia del generador fuera un `return` o un `throw`.

Recuerda que, si un iterador define un método `return()` y la iteración se detiene antes de tiempo, entonces el intérprete llama automáticamente al método `return()` para dar al iterador la oportunidad de cerrar los archivos o hacer otra limpieza. En el caso de los generadores, no se puede definir un método `return()` personalizado para manejar la limpieza, pero se puede estructurar el código del generador para usar una sentencia `try/finally` que asegure que se haga la limpieza necesaria (en el bloque `finally`) cuando el generador regrese. Al forzar el retorno del generador, el método `return()` incorporado en el generador asegura que el código de limpieza se ejecute cuando el generador ya no se utilice.

Al igual que el método `next()` de un generador nos permite pasar valores arbitrarios a un generador en ejecución, el método `throw()` de un generador nos da una forma de enviar señales arbitrarias (en forma de excepciones) a un generador. Llamar al método `throw()` siempre provoca una excepción dentro del generador. Pero si la función del generador está escrita con un código de

manejo de excepciones apropiado, la excepción no tiene por qué ser fatal, sino que puede ser un medio para alterar el comportamiento del generador. Imagínese, por ejemplo, un generador de contadores que produzca una secuencia creciente de enteros. Esto podría escribirse de manera que una excepción enviada con `throw()` pusiera el contador a cero.

Cuando un generador utiliza `yield*` para obtener valores de algún otro objeto iterable, una llamada al método `next()` del generador provoca una llamada al método `next()` del objeto iterable. Lo mismo ocurre con los métodos `return()` y `throw()`. Si un generador utiliza `yield*` en un objeto iterable que tiene definidos estos métodos, entonces la llamada a `return()` o `throw()` en el generador hace que el método `return()` o `throw()` del iterador sea llamado a su vez. Todos los iteradores *deben* tener un método `next()`. Los iteradores que necesitan limpiarse después de una iteración incompleta *deben* definir un método `return()`. Y cualquier iterador *puede definir* un método `throw()`, aunque no conozco ninguna razón práctica para hacerlo.

#### 12.4.4 Una nota final sobre los generadores

Los generadores son una estructura de control generalizada muy potente. Nos dan la capacidad de pausar una computación con rendimiento y reiniciarla de nuevo en algún momento posterior arbitrario con un valor de entrada arbitrario. Es posible utilizar los generadores para crear una especie de sistema de hilos cooperativos dentro del código JavaScript de un solo hilo. Y es posible usar generadores para enmascarar partes asíncronas de tu programa de manera que tu código parezca secuencial y síncrono,

aunque algunas de tus llamadas a funciones sean en realidad asíncronas y dependan de eventos de la red.

Intentar hacer estas cosas con generadores lleva a un código que es alucinantemente difícil de entender o explicar. Sin embargo, se ha hecho, y el único caso de uso realmente práctico ha sido para gestionar código asíncrono. Sin embargo, JavaScript tiene ahora las palabras clave `async` y `await` (véase [el capítulo 13](#)) para este mismo propósito, y ya no hay ninguna razón para abusar de los generadores de esta manera.

## 12.5 Resumen

En este capítulo has aprendido:

- El bucle `for/of` y el operador de propagación ... funcionan con objetos iterables.
- Un objeto es iterable si tiene un método con el nombre simbólico `[Symbol.iterator]` que devuelve un objeto iterador.
- Un objeto iterador tiene un método `next()` que devuelve un objeto resultado de la iteración.
- Un objeto resultado de una iteración tiene una propiedad `value` que contiene el siguiente valor iterado, si lo hay. Si la iteración se ha completado, entonces el objeto resultado debe tener una propiedad `done` establecida a `true`.
- Puedes implementar tus propios objetos iterables definiendo un método `[Symbol.iterator]()` que devuelva un objeto con un método `next()` que devuelva objetos resultado de la iteración. También

puedes implementar funciones que acepten argumentos de iteración y devuelvan valores de iteración.

- Las funciones generadoras (funciones definidas con `function*` en lugar de `function`) son otra forma de definir iteradores.
- Cuando se invoca una función generadora, el cuerpo de la función no se ejecuta inmediatamente; en su lugar, el valor de retorno es un objeto iterable iterador. Cada vez que se llama al método `next()` del iterador, se ejecuta otro trozo de la función generadora.
- Las funciones generadoras pueden utilizar el operador `yield` para especificar los valores que son devueltos por el iterador. Cada llamada a `next()` hace que la función generadora llegue hasta la siguiente expresión `yield`. El valor de esa expresión `yield` se convierte entonces en el valor devuelto por el iterador. Cuando no hay más expresiones `yield`, la función generadora regresa y la iteración se completa.



# Capítulo 13. JavaScript asíncrono

---

Algunos programas informáticos, como las simulaciones científicas y los modelos de aprendizaje automático, son de tipo computacional: se ejecutan continuamente, sin pausa, hasta que han calculado su resultado. Sin embargo, la mayoría de los programas informáticos del mundo real son significativamente *asíncronos*. Esto significa que a menudo tienen que dejar de computar mientras esperan que lleguen los datos o que ocurra algún evento. Los programas de JavaScript en un navegador web suelen estar *orientados a eventos*, lo que significa que esperan a que el usuario haga clic o toque antes de hacer algo. Y los servidores basados en JavaScript suelen esperar a que lleguen las peticiones de los clientes a través de la red antes de hacer nada.

Este tipo de programación asíncrona es habitual en JavaScript, y este capítulo documenta tres importantes características del lenguaje que ayudan a facilitar el trabajo con código asíncrono. Las promesas, nuevas en ES6, son objetos que representan el resultado aún no disponible de una operación asíncrona. Las palabras clave `async` y `await` se introdujeron en ES2017 y proporcionan una nueva sintaxis que simplifica la programación asíncrona al permitirle estructurar su código basado en promesas como si fuera síncrono. Por último, los iteradores asíncronos y el bucle `for/await` se introdujeron en ES2018 y

permiten trabajar con flujos de eventos asíncronos utilizando bucles simples que parecen síncronos.

Irónicamente, aunque JavaScript proporciona estas potentes características para trabajar con código asíncrono, no hay características del núcleo del lenguaje que sean en sí mismas asíncronas. Por lo tanto, para demostrar Promises, `async`, `await` y `for/await`, primero nos desviaremos hacia el JavaScript del lado del cliente y del lado del servidor para explicar algunas de las características asíncronas de los navegadores web y de Node. (Puedes aprender más sobre el JavaScript del lado del cliente y del lado del servidor en los capítulos [15](#) y [16](#)).

## 13.1 Programación asíncrona con callbacks

En su nivel más fundamental, la programación asíncrona en JavaScript se hace con *callbacks*. Una llamada de retorno es una función que se escribe y se pasa a otra función. Esa otra función entonces invoca ("devuelve la llamada") a tu función cuando se cumple alguna condición o se produce algún evento (asíncrono). La invocación de la función de devolución de llamada que proporcionas te notifica la condición o el evento, y a veces, la invocación incluirá argumentos de la función que proporcionan detalles adicionales. Esto es más fácil de entender con algunos ejemplos concretos, y las subsecciones que siguen demuestran varias formas de programación asíncrona basada en callbacks utilizando tanto JavaScript del lado del cliente como Node.

### 13.1.1 Temporizadores

Uno de los tipos más simples de asincronía es cuando se quiere ejecutar algún código después de que haya transcurrido una cierta cantidad de tiempo. Como vimos en §11.10, puedes hacer esto con la función `setTimeout()`:

```
setTimeout(checkForUpdates, 60000);
```

El primer argumento de `setTimeout()` es una función y el segundo es un intervalo de tiempo medido en milisegundos. En el código anterior, una hipotética función `checkForUpdates()` será llamada 60.000 milisegundos (1 minuto) después de la llamada a `setTimeout()`. `checkForUpdates()` es una función de llamada de retorno que su programa podría definir, y `setTimeout()` es la función que usted invoca para registrar su función de llamada de retorno y especificar bajo qué condiciones asíncronas debe ser invocada.

`setTimeout()` llama a la función callback especificada una vez, sin pasar argumentos, y luego se olvida de ella. Si está escribiendo una función que realmente comprueba las actualizaciones, probablemente quiera que se ejecute repetidamente. Puede hacer esto usando `setInterval()` en lugar de `setTimeout()`:

```
// Llama a checkForUpdates en un minuto y luego otra vez cada minuto después deja
updateIntervalId = setInterval(checkForUpdates, 60000);

// setInterval() devuelve un valor que podemos utilizar para detener la repetición
// invocaciones llamando a clearInterval(). (Del mismo modo, setTimeout() //
// devuelve un valor que puede pasar a clearTimeout()) function
stopCheckingForUpdates() { clearInterval(updateIntervalId);
}

}
```

### 13.1.2 Eventos

Los programas JavaScript del lado del cliente son casi universalmente impulsados por eventos: en lugar de ejecutar algún tipo de cálculo predeterminado, suelen esperar a que el usuario haga algo y luego responden a las acciones del usuario. El navegador web genera un *evento* cuando el usuario pulsa una tecla del teclado, mueve el ratón, hace clic en un botón del ratón o toca un dispositivo de pantalla táctil. Los programas JavaScript basados en eventos registran funciones de devolución de llamada para tipos de eventos específicos en contextos determinados, y el navegador web invoca esas funciones cada vez que se producen los eventos especificados. Estas funciones de devolución de llamada se llaman *manejadores de eventos* o *escuchadores de eventos*, y se registran con addEventListener():

```
// Pedir al navegador que devuelva un objeto que represente el
HTML
// Elemento <button> que coincide con este selector CSS let okay = document.
querySelector('#confirmUpdateDialog button.okay');

// Ahora registre una función callback para ser invocada cuando el usuario // haga
clic en ese botón.
okay.addEventListener('click', applyUpdate);
```

En este ejemplo, `applyUpdate()` es una hipotética función de devolución de llamada que suponemos está implementada en algún otro lugar. La llamada a `document.querySelector()` devuelve un objeto que representa un único elemento especificado en la página web. Llamamos a `addEventListener()` en ese elemento para registrar nuestra llamada de retorno. El primer argumento de `addEventListener()` es una cadena que especifica el tipo de evento que nos interesa: un clic del ratón o un toque de la pantalla táctil, en este caso. Si el usuario hace clic o toca en ese elemento específico de la página web, entonces el navegador invocará nuestra función de devolución de llamada `applyUpdate()`, pasando un objeto que incluye detalles (como la hora y las coordenadas del puntero del ratón) sobre el evento.

### 13.1.3 Eventos de red

Otra fuente común de asincronía en la programación de JavaScript son las peticiones de red. El JavaScript que se ejecuta en el navegador puede obtener datos de un servidor web con un código como este:

```
function getCurrentVersionNumber(versionCallback) { // Tenga en cuenta el  
argumento de la devolución de llamada // Realice una solicitud HTTP con script a una  
API de versión backend let request = new XMLHttpRequest(); request. open("GET",  
"http://www.example.com/api/version"); request. send();  
  
// Registra una llamada de retorno que será invocada cuando llegue la  
respuesta request. onload = function() { if (request. status === 200) {  
    // Si el estado HTTP es bueno, obtiene el número de versión y llama a la devolución  
    // de llamada.  
    let currentVersion = parseFloat(request. responseText);  
    versionCallback(null, currentVersion);  
} else { // De lo contrario, informa de un error a la devolución de llamada  
versionCallback(response. statusText, null); }  
};  
// Registrar otra llamada de retorno que será invocada para los errores de red  
request. onerror = request. ontimeout = function(e) { versionCallback(e. type,  
null); };  
}
```

El código JavaScript del lado del cliente puede utilizar la clase XMLHttpRequest más las funciones de devolución de llamada para realizar peticiones HTTP y manejar asíncronamente la respuesta del servidor cuando llegue.<sup>1</sup> El

La función getCurrentVersionNumber() definida aquí (podemos imaginar que es utilizada por la hipotética función checkForUpdates() que discutimos en §13.1.1) realiza una petición HTTP y define los manejadores de eventos que serán invocados cuando se reciba la

respuesta del servidor o cuando un tiempo de espera u otro error haga que la petición falle.

Observe que el código del ejemplo anterior no llama a `addEventListener()` como lo hizo nuestro ejemplo anterior. Para la mayoría de las APIs web (incluyendo ésta), los manejadores de eventos pueden ser definidos invocando `addEventListener()` en el objeto que genera el evento y pasando el nombre del evento de interés junto con la función callback. Sin embargo, normalmente también se puede registrar un único receptor de eventos asignándolo directamente a una propiedad del objeto. Eso es lo que hacemos en este código de ejemplo, asignando funciones a las propiedades `onload`, `onerror` y `ontimeout`. Por convención, las propiedades de escucha de eventos como éstas siempre tienen nombres que comienzan con `on`.

`addEventListener()` es la técnica más flexible porque permite múltiples manejadores de eventos. Pero en los casos en los que esté seguro de que ningún otro código necesitará registrar un oyente para el mismo objeto y tipo de evento, puede ser más sencillo simplemente establecer la propiedad apropiada a su devolución de llamada.

Otra cosa a tener en cuenta sobre la función `getCurrentVersionNumber()` en este código de ejemplo es que, debido a que hace una petición asíncrona, no puede devolver de forma sincrónica el valor (el número de la versión actual) en el que el llamador está interesado. En su lugar, la persona que llama pasa una función de devolución de llamada, que se invoca cuando el resultado está listo o cuando se produce un error. En este caso, la persona que

llama proporciona una función de devolución de llamada que espera dos argumentos. Si el XMLHttpRequest funciona correctamente, entonces getCurrentVersionNumber() invoca la llamada de retorno con un primer argumento nulo y el número de versión como segundo argumento. O, si se produce un error, entonces getCurrentVersionNumber() invoca la llamada de retorno con los detalles del error en el primer argumento y null como segundo argumento.

#### 13.1.4. Llamadas de retorno y eventos en Node

El entorno JavaScript del lado del servidor de Node.js es profundamente asíncrono y define muchas APIs que utilizan callbacks y eventos. La API por defecto para leer el contenido de un archivo, por ejemplo, es asíncrona e invoca una función de devolución de llamada cuando se ha leído el contenido del archivo:

```
const fs = require("fs"); // El módulo "fs" tiene APIs relacionadas con el sistema de
archivos let options = { // Un objeto para contener las opciones de nuestro programa
// las opciones por defecto irían aquí
};

// Leer un archivo de configuración, luego llamar a la función de devolución de llamada
fs.readFile("config.json", "utf-8", (err, text) => {
    // Si hay un error, muestra una advertencia, pero continúa console.
    warn("Could not read config file:", err);
} si no {
    // En caso contrario, analiza el contenido del archivo y lo asigna al objeto opciones
    Object.assign(options, JSON.parse(text));
}

}
```

```
// En cualquier caso, ya podemos empezar a ejecutar el programa
startProgram(options);});
```

La función `fs.readFile()` de Node toma un callback de dos parámetros como último argumento. Lee el archivo especificado de forma asíncrona y luego invoca la llamada de retorno. Si el archivo fue leído con éxito, pasa el contenido del archivo como el segundo argumento de la llamada de retorno. Si hubo un error, pasa el error como el primer argumento de la llamada de retorno. En este ejemplo, expresamos la llamada de retorno como una función de flecha, que es una sintaxis sucinta y natural para este tipo de operación simple.

Node también define una serie de APIs basadas en eventos. La siguiente función muestra cómo hacer una petición HTTP del contenido de una URL en Node. Tiene dos capas de código asíncrono manejadas con escuchadores de eventos. Fíjate en que Node utiliza un método `on()` para registrar oyentes de eventos en lugar de `addEventListener()`:

```
const https = require("https");

// Leer el contenido del texto de la URL y pasarlo de forma asíncrona a la llamada de retorno.
function getText(url, callback) { // Iniciar una petición HTTP GET
para la URL request = https.get(url);

    // Registra una función para manejar el evento "response". request.on("response",
response => {
    // El evento de respuesta significa que se han recibido las cabeceras de
respuesta let httpStatus = response.statusCode;

    // El cuerpo de la respuesta HTTP aún no se ha recibido.
```

```

// Así que registramos más manejadores de eventos para ser llamados cuando llegue.
response.setEncoding("utf-8"); // Esperamos texto Unicode let body = ""; // que
acumularemos aquí.

// Este manejador de eventos es llamado cuando un trozo del cuerpo está listo
respuesta.on("data", chunk => { body += chunk; });

// Este manejador de eventos es llamado cuando la respuesta está completa
response.on("end", () => { if (httpStatus === 200) { // Si la respuesta HTTP fue buena
callback(null, body); // Pasa el cuerpo de la respuesta al callback } else { // En caso
contrario pasa un error callback(httpStatus, null); }
});

});

// También registramos un manejador de eventos para los errores de red de
nivel inferior request.on("error", (err) => { callback(err, null); });
}

```

## 13.2 Promesas

Ahora que hemos visto ejemplos de programación asíncrona basada en devoluciones de llamada y en eventos en entornos JavaScript del lado del cliente y del lado del servidor, podemos introducir *Promises*, una característica central del lenguaje diseñada para simplificar la programación asíncrona.

Una promesa es un objeto que representa el resultado de un cálculo asíncrono. Ese resultado puede o no estar listo todavía, y la API de Promise es intencionadamente vaga al respecto: no hay forma de obtener de forma sincrónica el valor de una Promise; sólo se puede pedir a la Promise que llame a una función de devolución de llamada cuando el valor esté listo. Si estás definiendo una API asíncrona como la función `getText()` de la sección anterior, pero quieres que esté basada en una Promesa, omite el argumento de la llamada de retorno, y en su lugar devuelve un objeto Promise. La persona que llama puede entonces registrar una o más devoluciones de llamada en este objeto Promise, y serán invocadas cuando el cálculo asíncrono haya terminado.

Así que, en el nivel más simple, las promesas son sólo una forma diferente de trabajar con las devoluciones de llamada. Sin embargo, existen beneficios prácticos al usarlas. Un problema real con la programación asíncrona basada en callbacks es que es común terminar con callbacks dentro de callbacks dentro de callbacks, con líneas de código tan altamente sangradas que es difícil de leer. Las promesas permiten que este tipo de devolución de llamada anidada sea reexpresada como una *cadena de promesas* más lineal que tiende a ser más fácil de leer y más fácil de razonar.

Otro problema de las devoluciones de llamada es que pueden dificultar el manejo de errores. Si una función asíncrona (o una llamada de retorno invocada de forma asíncrona) lanza una excepción, no hay forma de que esa excepción se propague de vuelta al iniciador de la operación asíncrona. Este es un hecho fundamental

sobre la programación asíncrona: rompe el manejo de excepciones. La alternativa es rastrear y propagar meticulosamente los errores con argumentos de devolución de llamada y valores de retorno, pero esto es tedioso y difícil de hacer bien. Las promesas ayudan aquí estandarizando una forma de manejar los errores y proporcionando una manera de que los errores se propaguen correctamente a través de una cadena de promesas.

Tenga en cuenta que las promesas representan los resultados futuros de cálculos asíncronos individuales. Sin embargo, no pueden utilizarse para representar cálculos asíncronos repetidos. Más adelante en este capítulo, escribiremos una alternativa basada en Promise a la función setTimeout(), por ejemplo. Pero no podemos usar Promises para reemplazar setInterval() porque esa función invoca una función callback repetidamente, que es algo para lo que las Promises no están diseñadas. Del mismo modo, podríamos utilizar una Promise en lugar del manejador de eventos "load" de un objeto XMLHttpRequest, ya que esa llamada de retorno sólo se invoca una vez. Pero normalmente no usaríamos una Promise en lugar del manejador de eventos "click" de un objeto botón HTML, ya que normalmente queremos permitir que el usuario haga click en un botón varias veces.

Las subsecciones que siguen lo harán:

- Explicar la terminología de Promise y mostrar el uso básico de
- Promise
- Mostrar cómo se pueden encadenar las promesas

Demostrar cómo crear sus propias APIs basadas en Promise

### IMPORTANTE

Las promesas parecen sencillas al principio, y el caso de uso básico de las promesas es, de hecho, directo y sencillo. Pero pueden llegar a ser sorprendentemente confusas para cualquier cosa más allá de los casos de uso más simples. Las promesas son un poderoso lenguaje para la programación asíncrona, pero es necesario entenderlas profundamente para usarlas correctamente y con confianza. Sin embargo, vale la pena tomarse el tiempo para desarrollar esa comprensión profunda, y le insto a estudiar este largo capítulo cuidadosamente.

#### 13.2.1 Uso de promesas

Con la llegada de las promesas al núcleo del lenguaje JavaScript, los navegadores web han comenzado a implementar APIs basadas en promesas. En la sección anterior, implementamos una función `getText()` que realizaba una petición HTTP asíncrona y pasaba el cuerpo de la respuesta HTTP a una función callback especificada como una cadena. Imagina una variante de esta función, `getJSON()`, que analiza el cuerpo de la respuesta HTTP como JSON y devuelve una Promise en lugar de aceptar un argumento de callback.

Implementaremos una función `getJSON()` más adelante en este capítulo, pero por ahora, veamos cómo utilizaríamos esta función de utilidad que devuelve una Promesa:

```
getJSON(url).then(jsonData => {  
    // Esta es una función de devolución de llamada que será invocada //  
    // asincrónicamente con el valor JSON analizado cuando esté disponible.});
```

`getJSON()` inicia una petición HTTP asíncrona para la URL que especifiques y luego, mientras esa petición está pendiente, devuelve un objeto Promise. El objeto Promise define un método de instancia `then()`. En lugar de pasar nuestra función callback directamente a `getJSON()`, la pasamos al método `then()`. Cuando llega la respuesta

HTTP, el cuerpo de esa respuesta se analiza como JSON, y el valor analizado resultante se pasa a la función que pasamos a `then()`.

Puedes pensar en el método `then()` como un método de registro de llamadas de retorno como el método `addEventListener()` utilizado para registrar manejadores de eventos en el lado del cliente de JavaScript. Si llamas al método `then()` de un objeto Promise varias veces, se llamará a cada una de las funciones que especifiques cuando se complete el cómputo prometido.

Sin embargo, a diferencia de muchos escuchadores de eventos, una Promise representa un único cómputo, y cada función registrada con `then()` será invocada sólo una vez. Vale la pena señalar que la función que se pasa a `then()` se invoca de forma asíncrona, incluso si el cómputo asíncrono ya se ha completado cuando se llama a `then()`.

A un nivel sintáctico simple, el método `then()` es la característica distintiva de las promesas, y es idiomático añadir `.then()` directamente a la invocación de la función que devuelve la promesa, sin el paso intermedio de asignar el objeto promesa a una variable.

También es idiomático nombrar las funciones que devuelven promesas y las funciones que utilizan los resultados de las promesas con verbos, y estos modismos conducen a un código que es particularmente fácil de leer:

```
// Suponga que tiene una función como esta para mostrar un perfil de usuario
function displayUserProfile(profile) { /* implementation omitted */}

// Así es como podrías usar esa función con una Promesa.
// Fíjate en que esta línea de código se lee casi como una frase en inglés:
getJSON("/api/user/profile"). then(displayUserProfile);
```

## MANEJO DE ERRORES CON PROMESAS

Las operaciones asíncronas, en particular las que implican la creación de redes, suelen fallar de varias maneras, y hay que escribir un código robusto para manejar los errores que inevitablemente se producirán.

En el caso de las promesas, podemos hacerlo pasando una segunda función al método `then()`:

```
getJSON("/api/user/profile"). then(displayUserProfile, handleProfileError);
```

Una Promise representa el resultado futuro de una computación asíncrona que ocurre después de la creación del objeto Promise. Dado que el cálculo se realiza después de que el objeto Promise nos sea devuelto, no hay forma de que el cálculo pueda devolver tradicionalmente un valor o lanzar una excepción que podamos atrapar. Las funciones que pasamos a `then()` proporcionan alternativas. Cuando una computación sincrónica se completa normalmente, simplemente devuelve su resultado a su convocante. Cuando una computación asíncrona basada en promesas se completa normalmente, pasa su resultado a la función que es el primer argumento de `then()`.

Cuando algo va mal en una computación sincrónica, se lanza una excepción que se propaga por la pila de llamadas hasta que hay una cláusula catch para manejarla. Cuando una computación asíncrona se ejecuta, su llamador ya no está en la pila, por lo que si algo va mal, simplemente no es posible lanzar una excepción de vuelta al llamador.

En su lugar, los cálculos asíncronos basados en promesas pasan la excepción (normalmente como un objeto Error de algún tipo, aunque esto no es necesario) a la segunda función pasada a then(). Así, en el código anterior, si getJSON() se ejecuta normalmente, pasa su resultado a displayUserProfile(). Si hay un error (el usuario no ha iniciado la sesión, el servidor está caído, la conexión a Internet del usuario se ha caído, la solicitud ha expirado, etc.), entonces getJSON() pasa un objeto Error a handleProfileError().

En la práctica, es raro ver dos funciones pasadas a then(). Hay una forma mejor y más idiomática de manejar los errores cuando se trabaja con Promesas. Para entenderlo, primero considere lo que sucede si getJSON() se completa normalmente pero se produce un error en displayUserProfile(). Esa función de devolución de llamada es invocada de forma asíncrona cuando getJSON() regresa, por lo que también es asíncrona y no puede lanzar una excepción de forma significativa (porque no hay código en la pila de llamadas para manejarla).

La forma más idiomática de manejar los errores en este código es la siguiente:

```
getJSON("/api/user/profile").then(displayUserProfile).catch(handleProfileError);
```

Con este código, un resultado normal de `getJSON()` se sigue pasando a `displayUserProfile()`, pero cualquier error en `getJSON()` o en `displayUserProfile()` (incluyendo cualquier excepción lanzada por `displayUserProfile()`) se pasa a `handleProfileError()`. El método `catch()` es sólo una forma abreviada de llamar a `then()` con un primer argumento nulo y la función manejadora de errores especificada como segundo argumento.

Tendremos más que decir sobre `catch()` y este lenguaje de gestión de errores cuando hablemos de las cadenas Promise en la siguiente sección.

## TERMINOLOGÍA DE LA

Antes Cuando hablamos de las promesas, vale la pena detenerse a definir algunos términos. Cuando no estamos programación y hablamos de promesas humanas, decimos que una promesa se "cumple" o se "rompe". Cuando hablamos de promesas en JavaScript, los términos equivalentes son "cumplida" y "rechazada". Imagina que has llamado al método `then()` de una promesa y le has pasado dos funciones de devolución de llamada. Decimos que la promesa se ha *cumplido* si y cuando se llama a la primera llamada de retorno. Y decimos que la promesa ha sido *rechazada si y cuando se llama a la segunda llamada de retorno*. Si una promesa no se cumple ni se rechaza, entonces está *pendiente*. Y una vez que una promesa se cumple o se rechaza, decimos que está *resuelta*. Ten en cuenta que una promesa nunca puede ser ni cumplida *ni rechazada*. Una vez que una promesa se resuelve, nunca cambiará de cumplida a rechazada o viceversa.

Recuerda cómo hemos definido las Promesas al principio de esta sección: "una Promesa es un objeto que representa el *resultado* de una operación asíncrona". Es importante recordar que las Promesas no son sólo formas abstractas de registrar callbacks para que se ejecuten cuando algún código asíncrono termine: representan los resultados de ese código asíncrono. Si el código asíncrono se ejecuta normalmente (y la Promesa se cumple), entonces ese resultado es esencialmente el valor de retorno del código. Y si el código asíncrono no se completa normalmente (y la Promesa es rechazada), entonces el resultado es un objeto Error o algún otro valor que el código podría haber lanzado si no fuera asíncrono. Cualquier Promise que se haya cumplido tiene un valor asociado, y ese valor no cambiará. Si la Promesa se cumple, entonces el valor es un valor de retorno que se pasa a cualquier función de devolución de llamada registrada como primer argumento de `then()`. Si la Promesa es rechazada, entonces el valor es un error de algún tipo que se pasa a cualquier función de devolución de llamada registrada con `catch()` o como el segundo argumento de `then()`.

La razón por la que quiero ser preciso sobre la terminología de las promesas es que éstas también pueden *resolverse*. Es fácil confundir este estado resuelto con el estado cumplido o con el estado resuelto, pero no es precisamente lo mismo que ninguno de los dos. Entender el estado resuelto es una de las claves para comprender en profundidad las Promesas, y volveré a ello después de haber hablado de las cadenas de Promesas más adelante.

### 13.2.2 Encadenamiento de promesas

Una de las ventajas más importantes de las promesas es que proporcionan una forma natural de expresar una secuencia de operaciones asíncronas como una cadena lineal de invocaciones al método `then()`, sin tener que anidar cada operación dentro de la llamada de retorno de la anterior. Aquí, por ejemplo, hay una hipotética cadena de Promesas:

```
fetch(documentURL) // Hacer una petición HTTP . then(response => response.json())
// Pedir el cuerpo JSON de la respuesta . then(document => { // Cuando obtenemos el
JSON analizado return render(document); // mostrar el documento al usuario
})
. then(rendered => { // Cuando obtenemos el documento renderizado
cacheInDatabase(rendered); // lo almacenamos en la base de datos local. })
. catch(error => handle(error)); // Manejar cualquier error que ocurra
```

Este código ilustra cómo una cadena de Promises puede facilitar la expresión de una secuencia de operaciones asíncronas. Sin embargo, no vamos a discutir esta cadena Promise en particular. Sin embargo, seguiremos explorando la idea de utilizar cadenas Promise para realizar peticiones HTTP.

Anteriormente en este capítulo, vimos el objeto XMLHttpRequest utilizado para hacer una petición HTTP en JavaScript. Ese objeto de nombre extraño tiene una API antigua y complicada, y ha sido reemplazada en gran medida por la nueva API Fetch, basada en promesas ([§15.11.1](#)). En su forma más simple, esta nueva API HTTP es sólo la función fetch(). Se le pasa una URL y devuelve una promesa. Esa promesa se cumple cuando la respuesta HTTP comienza a llegar y el estado y las cabeceras HTTP están disponibles:

```
fetch("/api/user/profile").then(response => { // Cuando la promesa se resuelve,  
tenemos el estado y las cabeceras if (response.ok && response.headers.  
get("Content-Type") ===  
"application/json") {  
    // ¿Qué podemos hacer aquí? Todavía no tenemos el cuerpo de la respuesta.  
}  
});
```

Cuando la Promesa devuelta por fetch() se cumple, pasa un objeto Response a la función que pasaste a su método then(). Este objeto response te da acceso al estado de la solicitud y a las cabeceras, y también define métodos como text() y json(), que te dan acceso al cuerpo de la respuesta en forma de texto y JSON, respectivamente. Pero aunque la Promesa inicial se cumpla, el cuerpo de la respuesta puede no haber llegado todavía. Así que estos métodos text() y json() para acceder al cuerpo de la respuesta devuelven ellos mismos Promesas. He aquí una forma ingenua de utilizar fetch() y el método response.json() para obtener el cuerpo de una respuesta HTTP:

```
fetch("/api/user/profile").then(response => { response.json().then(profile => { // Pedir el cuerpo parseado en JSON
    // Cuando el cuerpo de la respuesta llegue, será automáticamente // parseado como JSON y pasado a esta función.
    displayUserProfile(profile); });
});
```

Esta es una forma ingenua de usar las promesas porque las anidamos, como si fueran callbacks, lo que no tiene sentido. El lenguaje preferido es usar Promesas en una cadena secuencial con código como este:

```
fetch("/api/user/profile") . then(response =>
{ return response.json(); }) . then(profile =>
{
    mostrarPerfilDeUsuario(perfil);
});
```

Veamos las invocaciones de métodos en este código, ignorando los argumentos que se pasan a los métodos:

```
fetch().then().then()
```

Cuando se invoca más de un método en una sola expresión como ésta, la llamamos *cadena de métodos*. Sabemos que la función `fetch()` devuelve un objeto `Promise`, y podemos ver que el primer `.then()` de esta cadena invoca un método sobre ese objeto `Promise` devuelto. Pero hay un segundo `.then()` en la cadena, lo que significa que la primera invocación del método `then()` debe devolver a su vez una `Promise`.

A veces, cuando una API está diseñada para utilizar este tipo de encadenamiento de métodos, sólo hay un único objeto, y cada método de ese objeto devuelve el propio objeto para facilitar el encadenamiento. Sin embargo, no es así como funcionan las promesas. Cuando escribimos una cadena de invocaciones .then(), no estamos registrando múltiples devoluciones de llamada en un único objeto Promise. En su lugar, cada invocación del método then() devuelve un nuevo objeto Promise. Ese nuevo objeto Promise no se cumple hasta que la función pasada a then() se completa.

Volvamos a una forma simplificada de la cadena original de fetch() anterior. Si definimos las funciones pasadas a las invocaciones de then() en otro lugar, podríamos refactorizar el código para que se vea así:

```
fetch(theURL) // tarea 1; devuelve la promesa 1 . then(callback1) // tarea 2; devuelve la  
promesa 2  
. then(callback2); // tarea 3; devuelve la promesa 3
```

Veamos este código en detalle:

1. En la primera línea, fetch() se invoca con una URL. Inicia una petición HTTP GET para esa URL y devuelve una promesa. Llamaremos a esta petición HTTP "tarea 1" y llamaremos a la Promesa "promesa 1".
2. En la segunda línea, invocamos el método then() de la promesa 1, pasando la función callback1 que queremos que se invoque cuando se cumpla la promesa 1. El método then() almacena nuestra llamada de retorno en algún lugar, y luego

devuelve una nueva promesa. Llamaremos a la nueva Promise devuelta en este paso "promesa 2", y diremos que la "tarea 2" comienza cuando se invoca el callback1.

3. En la tercera línea, invocamos el método then() de la promesa 2, pasando la función callback2 que queremos que se invoque cuando se cumpla la promesa 2. Este método then() recuerda nuestra devolución de llamada y devuelve otra promesa. Diremos que la "tarea 3" comienza cuando se invoca el callback2. Podemos llamar a esta última Promise "promesa 3", pero realmente no necesitamos un nombre para ella porque no la usaremos en absoluto.
4. Los tres pasos anteriores ocurren de forma sincrónica cuando la expresión se ejecuta por primera vez. Ahora tenemos una pausa asíncrona mientras la petición HTTP iniciada en el paso 1 se envía a través de Internet.
5. Finalmente, la respuesta HTTP comienza a llegar. La parte asíncrona de la llamada fetch() envuelve el estado HTTP y las cabeceras en un objeto Response y cumple la promesa 1 con ese objeto Response como valor.
6. Cuando se cumple la promesa 1, su valor (el objeto Response) se pasa a nuestra función callback1(), y comienza la tarea 2. El trabajo de esta tarea, dado un objeto Response como entrada, es obtener el cuerpo de la respuesta como un objeto JSON.
7. Supongamos que la tarea 2 se completa normalmente y es capaz de analizar el cuerpo de la respuesta HTTP para producir un objeto JSON. Este objeto JSON se utiliza para cumplir la promesa 2.

8. El valor que cumple la promesa 2 se convierte en la entrada de la tarea 3 cuando se pasa a la función callback2(). Esta tercera tarea muestra ahora los datos al usuario de alguna manera no especificada. Cuando la tarea 3 se complete (asumiendo que se complete normalmente), entonces la promesa 3 se cumplirá. Pero como nunca hicimos nada con la promesa 3, no pasa nada cuando esa promesa se cumple, y la cadena de computación asíncrona termina en este punto.

### 13.2.3 Resolver las promesas

Mientras explicábamos la cadena de promesas de obtención de URL con la lista en la última sección, hablamos de las promesas 1, 2 y 3. Pero en realidad hay un cuarto objeto Promise involucrado también, y esto nos lleva a nuestra importante discusión sobre lo que significa que una Promise sea "resuelta".

Recuerda que `fetch()` devuelve un objeto Promise que, cuando se cumple, pasa un objeto Response a la función callback que registramos. Este objeto Response tiene métodos `.text()`, `.json()`, y otros métodos para solicitar el cuerpo de la respuesta HTTP en varias formas. Pero como el cuerpo puede no haber llegado todavía, estos métodos deben devolver objetos Promise. En el ejemplo que hemos estado estudiando, la "tarea 2" llama al método `.json()` y devuelve su valor. Este es el cuarto objeto Promise, y es el valor de retorno de la función `callback1()`.

Volvamos a escribir el código de obtención de la URL una vez más de una manera verbosa y no idiomática que haga explícitas las devoluciones de llamada y las promesas:

```
function c1(response) { // callback 1
  let p4 = response.json(); return p4; // devuelve la promesa 4 }

function c2(profile) { // callback 2 displayUserProfile(profile); }

let p1 = fetch("/api/user/profile"); // promesa 1, tarea 1 let p2 = p1.then(c1); // promesa 2, tarea 2 let p3 = p2.then(c2); // promesa 3, tarea 3
```

Para que las cadenas Promise funcionen de forma útil, la salida de la tarea 2 debe convertirse en la entrada de la tarea 3. Y en el ejemplo que estamos considerando aquí, la entrada de la tarea 3 es el cuerpo de la URL que se obtuvo, analizado como un objeto JSON. Pero, como acabamos de discutir, el valor de retorno de la llamada de retorno c1 no es un objeto JSON, sino la Promesa p4 para ese objeto JSON. Esto parece una contradicción, pero no lo es: cuando se cumple p1, se invoca c1, y comienza la tarea 2. Y cuando se cumple p2, se invoca c2 y comienza la tarea 3. Pero el hecho de que la tarea 2 comience cuando se invoca a c1 no significa que la tarea 2 deba terminar cuando c1 regrese. Después de todo, las promesas son para gestionar tareas asíncronas, y si la tarea 2 es asíncrona (que lo es, en este caso), entonces esa tarea no estará completa cuando regrese la llamada de retorno.

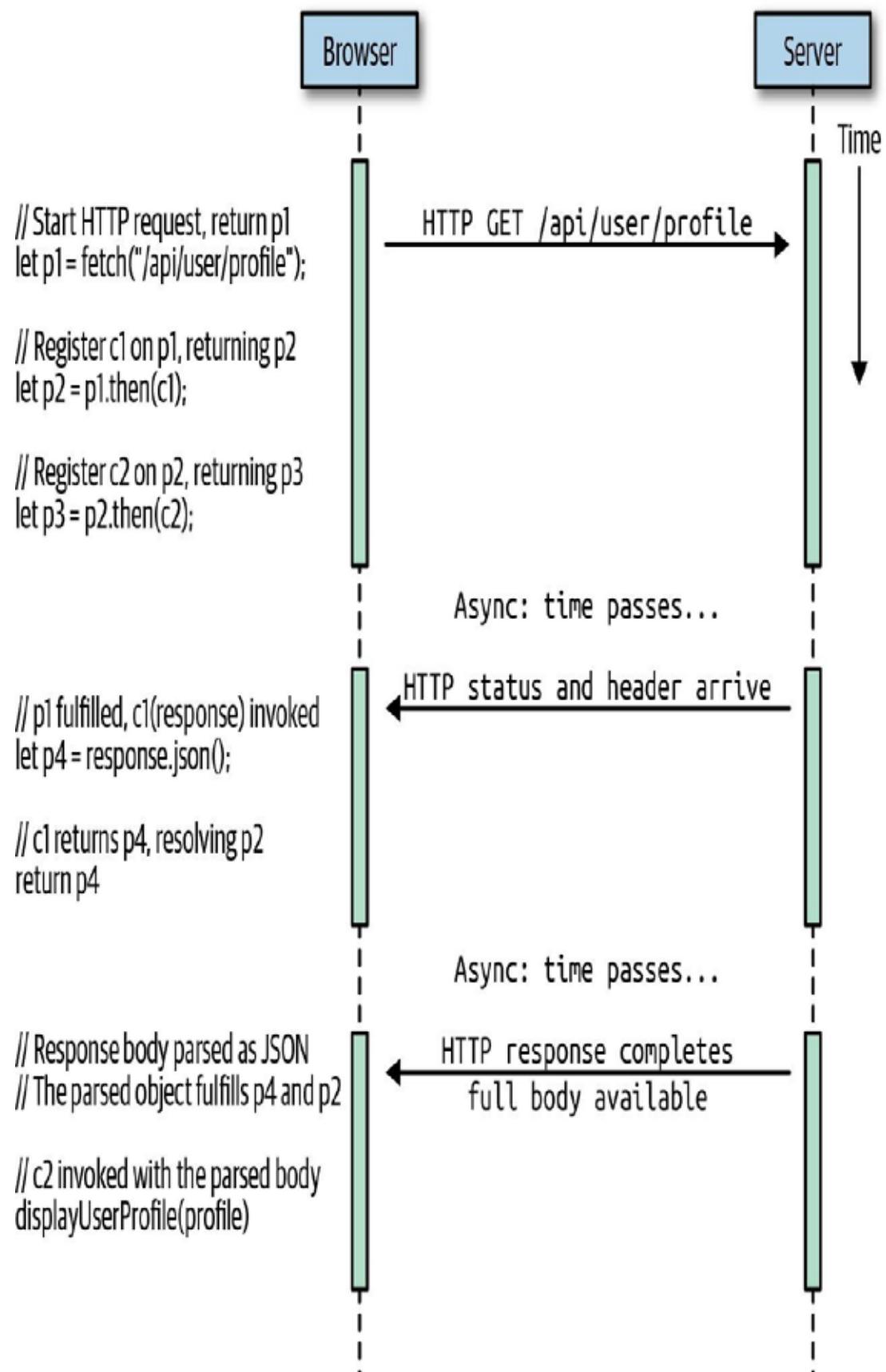
Ahora estamos listos para discutir el detalle final que necesitas entender para dominar realmente las Promesas. Cuando pasas una llamada de retorno c al método then(), then() devuelve una Promesa p y se encarga de invocar asíncronamente a c en algún momento

posterior. La llamada de retorno realiza algún cálculo y devuelve un valor v. Cuando la llamada de retorno regresa, p se *resuelve* con el valor v. Cuando una Promise se resuelve con un valor que no es en sí mismo una Promise, se cumple inmediatamente con ese valor. Así que si c devuelve un valor que no es una Promise, ese valor devuelto se convierte en el valor de p, p se cumple y ya hemos terminado. Pero si el valor de retorno v es en sí mismo una Promise, entonces p se resuelve *pero no se cumple todavía*. En esta etapa, p no puede resolverse hasta que se cumpla la Promesa v. Si v se cumple, entonces p se cumplirá con el mismo valor. Si v se rechaza, entonces p se rechazará por la misma razón. Esto es lo que significa el estado "resuelto" de una Promesa: la Promesa se ha asociado con, o "bloqueado", otra Promesa. Todavía no sabemos si p se cumplirá o será rechazada, pero nuestra llamada de retorno c ya no tiene ningún control sobre eso. p está "resuelta" en el sentido de que su destino ahora depende completamente de lo que ocurra con la promesa v.

Volvamos a nuestro ejemplo de obtención de URLs. Cuando c1 devuelve p4, p2 se resuelve. Pero estar resuelto no es lo mismo que estar cumplido, así que la tarea 3 no comienza todavía. Cuando el cuerpo completo de la respuesta HTTP esté disponible, el método .json() podrá analizarlo y utilizar ese valor analizado para cumplir con p4. Cuando p4 se cumple, p2 se cumple automáticamente también, con el mismo valor JSON analizado. En este punto, el objeto JSON analizado se pasa a c2, y comienza la tarea 3.

Esta puede ser una de las partes más difíciles de entender de JavaScript, y es posible que tenga que leer esta sección más de una

vez. La Figura 13-1 presenta el proceso de forma visual y puede ayudar a aclararlo.



*Figura 13-1. Obtención de una URL con promesas*

### 13.2.4 Más sobre las promesas y los errores

Anteriormente en el capítulo, vimos que puedes pasar una segunda función callback al método `.then()` y que esta segunda función será invocada si la Promesa es rechazada. Cuando esto sucede, el argumento de esta segunda función de llamada de retorno es un valor -típicamente un objeto Error- que representa la razón del rechazo. También hemos aprendido que es poco común (e incluso poco idiomático) pasar dos llamadas de retorno a un método `.then()`. En su lugar, los errores relacionados con las promesas se gestionan normalmente añadiendo una invocación al método `.catch()` a una cadena de promesas. Ahora que hemos examinado las cadenas Promise, podemos volver a la gestión de errores y discutirla con más detalle. Como preámbulo a la discusión, me gustaría destacar que el manejo cuidadoso de los errores es realmente importante cuando se hace programación asíncrona. Con el código síncrono, si dejas de lado el código de manejo de errores, al menos obtendrás una excepción y un seguimiento de la pila que puedes usar para averiguar qué está fallando. Con el código asíncrono, las excepciones no manejadas a menudo no serán reportadas, y los errores pueden ocurrir silenciosamente, haciéndolos mucho más difíciles de depurar. La buena noticia es que el método `.catch()` facilita el manejo de errores cuando se trabaja con promesas.

#### LOS MÉTODOS CATCH Y FINALLY

El método `.catch()` de una Promise es simplemente una forma abreviada de llamar a `.then()` con null como primer argumento y una llamada de retorno para el manejo de errores como segundo

argumento. Dada una Promesa p y una llamada de retorno c, las siguientes dos líneas son equivalentes:

```
p. then(null, c);  
p. coger (c);
```

La abreviatura .catch() se prefiere porque es más simple y porque el nombre coincide con la cláusula catch en una declaración de manejo de excepciones try/catch. Como hemos comentado, las excepciones normales no funcionan con código asíncrono. El método .catch() de Promises es una alternativa que sí funciona para el código asíncrono. Cuando algo va mal en el código síncrono, podemos hablar de una excepción que "sube por la pila de llamadas" hasta que encuentra un bloque catch. Con una cadena asíncrona de promesas, la metáfora comparable podría ser la de un error "goteando por la cadena" hasta que encuentra una invocación .catch().

En ES2018, los objetos Promise también definen un método .finally() cuyo propósito es similar a la cláusula finally en una sentencia try/catch/finally. Si añades una invocación .finally() a tu cadena de Promise, la llamada de retorno que pases a .finally() se invocará cuando la Promise sobre la que la llamaste se cumpla. Tu callback será invocado si la Promesa se cumple o se rechaza, y no se le pasará ningún argumento, por lo que no podrás averiguar si se cumplió o se rechazó. Pero si necesitas ejecutar algún tipo de código de limpieza (como cerrar archivos abiertos o conexiones de red) en cualquiera de los dos casos, una llamada de retorno .finally() es la forma ideal de hacerlo. Al igual que .then() y .catch(), .finally() devuelve un nuevo objeto Promise. El valor de retorno de una llamada de retorno .finally() generalmente se ignora, y la promesa devuelta por .finally()

normalmente se resolverá o rechazará con el mismo valor con el que se resuelve o rechaza la promesa sobre la que se invocó `.finally()`. Sin embargo, si una llamada de retorno `.finally()` lanza una excepción, la promesa devuelta por `.finally()` se rechazará con ese valor.

El código de obtención de URLs que hemos estudiado en las secciones anteriores no hacía ninguna gestión de errores. Vamos a corregirlo ahora con una versión más realista del código:

```
fetch("/api/user/profile") // Iniciar la petición HTTP
  .then(response => { // Llama a esto cuando el estado y las cabeceras están listos if (!
    response.ok) { // Si tenemos un error 404 Not Found o similar return null; // Tal vez el
      usuario ha cerrado la sesión; return null profile }
  )

  // Ahora comprueba las cabeceras para asegurarte de que el servidor nos ha
  enviado JSON.      // Si no es así, nuestro servidor está roto, y esto es un error
  grave!
  let type = response.headers.get("content-type"); if (type !==
  "application/json") { throw new TypeError(`Expected JSON, got ${type}`);
  }

  // Si llegamos aquí, entonces tenemos un estado 2xx y un
  Tipo de contenido JSON
  // para que podamos devolver con seguridad una Promise para el cuerpo de
  la respuesta // como objeto JSON. return response.json(); })
  .then(profile => { // Se llama con el cuerpo de la respuesta analizada o null if
  (profile) { displayUserProfile(profile);
  }
```

```
else { // Si tenemos un error 404 arriba y devolvemos null terminamos aquí
  displayLoggedOutProfilePage(); }

}) . catch(e => { if (e instanceof NetworkError) {

  //fetch() puede fallar de esta manera si la conexión a internet está caída
  displayErrorMessage("Compruebe su conexión a Internet."); } else if (e
instanceof TypeError) { // Esto ocurre si lanzamos TypeError por encima de
  displayErrorMessage("¡Algo va mal con nuestro servidor!"); } else { // Esto debe ser
  algún tipo de error imprevisto console.error(e); }

});
```

Analicemos este código observando lo que ocurre cuando las cosas van mal. Utilizaremos el esquema de nomenclatura que usamos antes: p1 es la promesa devuelta por la llamada fetch(). p2 es la promesa devuelta por la primera llamada .then(), y c1 es la llamada de retorno que pasamos a esa llamada .then(). p3 es la promesa devuelta por la segunda llamada .then(), y c2 es la llamada de retorno que pasamos a esa llamada. Finalmente, c3 es la llamada de retorno que pasamos a la llamada .catch(). (Esa llamada devuelve una Promise, pero no necesitamos referirnos a ella por su nombre).

Lo primero que podría fallar es la propia petición fetch(). Si la conexión de red está caída (o por alguna otra razón no se puede hacer una petición HTTP), entonces la promesa p1 será rechazada con un objeto NetworkError. No pasamos una función de devolución de llamada de manejo de errores como segundo

argumento a la llamada `.then()`, por lo que `p2` también se rechaza con el mismo objeto `NetworkError`. (Si hubiéramos pasado un manejador de errores a esa primera llamada `.then()`, el manejador de errores sería invocado, y si regresara normalmente, `p2` sería resuelto y/o cumplido con el valor de retorno de ese manejador). Sin un manejador, sin embargo, `p2` es rechazado, y luego `p3` es rechazado por la misma razón. En este punto, se llama a la llamada de retorno del manejador de errores `c3`, y se ejecuta el código específico de `NetworkError` dentro de ella.

Otra forma en que nuestro código podría fallar es si nuestra petición HTTP devuelve un 404 Not Found u otro error HTTP. Estas son respuestas HTTP válidas, por lo que la llamada a `fetch()` no las considera errores. `fetch()` encapsula un 404 Not Found en un objeto `Response` y llena `p1` con ese objeto, haciendo que se invoque `c1`. Nuestro código en `c1` comprueba la propiedad `ok` del objeto `Response` para detectar que no ha recibido una respuesta HTTP normal y maneja ese caso simplemente devolviendo `null`. Como este valor de retorno no es una `Promise`, cumple con `p2` de inmediato, y `c2` es invocado con este valor. Nuestro código en `c2` comprueba explícitamente y maneja los valores falsos mostrando un resultado diferente al usuario. Este es un caso en el que tratamos una condición anormal como un no-error y lo manejamos sin usar realmente un manejador de errores.

Un error más grave ocurre en `c1` si obtenemos un código de respuesta HTTP normal pero la cabecera `Content-Type` no está configurada adecuadamente. Nuestro código espera una respuesta con formato JSON, así que si el servidor nos envía HTML, XML o texto plano en su lugar, vamos a tener un problema. `c1` incluye

código para comprobar la cabecera Content-Type. Si la cabecera es incorrecta, trata esto como un problema no recuperable y lanza un TypeError. Cuando una llamada de retorno pasada a .then() (o a .catch()) arroja un valor, la promesa que era el valor de retorno de la llamada a .then() es rechazada con ese valor arrojado. En este caso, el código en c1 que lanza un TypeError hace que p2 sea rechazado con ese objeto TypeError. Como no especificamos un manejador de errores para p2, p3 será rechazado también. c2 no será llamado, y el TypeError será pasado a c3, que tiene código para comprobar explícitamente y manejar este tipo de error.

Hay un par de cosas que vale la pena señalar sobre este código. En primer lugar, fíjate en que el objeto de error lanzado con una sentencia throw normal y sincrónica acaba siendo manejado de forma asíncrona con una invocación al método .catch() en una cadena Promise. Esto debería dejar claro por qué este método abreviado es preferible a pasar un segundo argumento a .then(), y también por qué es tan idiomático terminar las cadenas Promise con una llamada a .catch().

Antes de dejar el tema del manejo de errores, quiero señalar que, aunque es idiomático terminar cada cadena Promise con un .catch() para limpiar (o al menos registrar) cualquier error que haya ocurrido en la cadena, también es perfectamente válido usar .catch() en cualquier otra parte de una cadena Promise. Si una de las etapas de tu cadena Promise puede fallar con un error, y si el error es algún tipo de error recuperable que no debería impedir que el resto de la cadena se ejecute, entonces puedes insertar una llamada .catch() en la cadena, resultando en un código que podría ser como este

```
startAsyncOperation() .then(doStageTwo)
```

```
. catch(recoverFromStageTwoError)
. entonces(doStageThree)
. entonces(doStageFour)
. catch(logStageThreeAndFourErrors);
```

Recuerde que la llamada de retorno que pase a .catch() sólo será invocada si la llamada de retorno en una etapa anterior arroja un error. Si la llamada de retorno retorna normalmente, entonces la llamada de retorno .catch() será omitida, y el valor de retorno de la llamada de retorno anterior se convertirá en la entrada de la siguiente llamada de retorno .then(). Recuerde también que las retrolllamadas .catch() no sólo sirven para informar de los errores, sino para manejarlos y recuperarse de ellos. Una vez que un error ha sido pasado a una llamada de retorno .catch(), deja de propagarse por la cadena Promise. Una llamada de retorno .catch() puede lanzar un nuevo error, pero si regresa normalmente, ese valor de retorno se utiliza para resolver y/o cumplir la Promesa asociada, y el error deja de propagarse.

Seamos concretos: en el ejemplo de código anterior, si startAsyncOperation() o doStageTwo() arroja un error, entonces se invocará la función recoverFromStageTwoError(). Si recoverFromStageTwoError() devuelve normalmente, su valor de retorno se pasará a doStageThree() y la operación asíncrona continuará normalmente. Por otro lado, si recoverFromStageTwoError() no ha podido recuperarse, lo hará lanzará un error (o volverá a lanzar el error que se le pasó). En este caso, ni doStageThree() ni doStageFour() serán invocados, y el error lanzado por recoverFromStageTwoError() se pasaría a logStageThreeAndFourErrors().

A veces, en entornos de red complejos, los errores pueden producirse de forma más o menos aleatoria, y puede ser conveniente manejar esos errores simplemente reintentando la petición asíncrona. Imagina que has escrito una operación basada en Promise para consultar una base de datos:

```
queryDatabase() . then(displayTable)  
  . catch(displayDatabaseError);
```

Ahora supongamos que los problemas de carga transitoria de la red están causando que esto falle alrededor del 1% de las veces. Una solución sencilla podría ser reintentar la consulta con una llamada `.catch()`:

```
queryDatabase()  
  . catch(e => wait(500). then(queryDatabase)) // En caso de fallo, esperar y  
  reintentar . then(displayTable)  
  . catch(displayDatabaseError);
```

Si los fallos hipotéticos son realmente aleatorios, añadir esta única línea de código debería reducir su tasa de error del 1% al 0,01%.

#### QUE REGRESA DE UNA PROMESA DE DEVOLUCIÓN DE LLAMADA

Volvamos por última vez al ejemplo anterior de obtención de URL, y consideremos la llamada de retorno `c1` que pasamos a la primera invocación de `.then()`. Observe que hay tres maneras en que `c1` puede terminar. Puede regresar normalmente con la Promesa devuelta por la llamada `.json()`. Esto hace que `p2` se resuelva, pero si esa Promesa se cumple o se rechaza depende de lo que ocurra con la nueva Promesa devuelta. `c1` también puede regresar normalmente con el valor `null`, lo que hace que `p2` se cumpla inmediatamente. Por último, `c1` puede terminar lanzando un error, lo que hace que `p2` sea rechazado. Estos son los tres resultados posibles para una Promesa, y el código en `c1` demuestra cómo la llamada de retorno puede causar cada resultado.

En una cadena Promise, el valor devuelto (o lanzado) en una etapa de la cadena se convierte en la entrada de la siguiente etapa de la cadena, por lo que es fundamental hacerlo bien. En la práctica, olvidarse de devolver un valor desde una función de devolución de llamada es en realidad una fuente común de errores relacionados con las promesas, y esto se ve exacerbado por la sintaxis de atajo de la función de flecha de JavaScript. Considere esta línea de código que vimos antes:

```
.atrap(e => esper(500).ento(queryDatabase))
```

Recuerdo de Capítulo 8 que las funciones de flecha permiten muchos atajos. Como hay exactamente una (el valor del error), podemos omitir los paréntesis. Dado que el cuerpo de la función es un único podemos omitir las llaves alrededor del cuerpo de la función y el valor de la expresión se convierte en el valor de retorno de la función. Gracias a estos atajos, el código anterior es correcto. Pero considere este cambio que parece inocuo:

```
.atrap(e => { esper(500).ento(queryDatabase) })
```

Al añadir las llaves, ya no obtenemos el retorno automático. Esta función ahora devuelve `indefinido` en lugar de devolver una Promise, lo que significa que la siguiente etapa de esta cadena Promise será ser invocado `indefinido` como su entrada en lugar del resultado de la consulta reintentada. Es un error sutil que puede no ser fácil de depurar.

### 13.2.5 Promesas en paralelo

Hemos pasado mucho tiempo hablando de las cadenas Promise para ejecutar secuencialmente los pasos asíncronos de una operación asíncrona mayor. Sin embargo, a veces queremos ejecutar una serie de operaciones asíncronas en paralelo. La función `Promise.all()` puede hacer esto. `Promise.all()` toma un array de objetos Promise como entrada y devuelve una Promise. La promesa devuelta será rechazada si alguna de las promesas de entrada es rechazada. En caso contrario, se cumplirá con un array de los valores de cumplimiento de cada una de las Promesas de entrada. Así que, por ejemplo, si quieras obtener el contenido de texto de múltiples URLs, podrías usar un código como este

```
// Empezamos con un array de URLs const urls = [ /* cero o más URLs aquí */ ]; // Y lo convertimos en un array de objetos Promise promises = urls.map(url => fetch(url).then(r => r.text())); // Ahora obtenemos una Promise para ejecutar todas esas Promises en paralelo Promise.all(promises) . then(bodies => { /* hacer algo con la matriz de cadenas */ })
```

```
.catch(e => console.error(e));
```

Promise.all() es ligeramente más flexible que lo descrito anteriormente. El array de entrada puede contener tanto objetos Promise como valores no Promise. Si un elemento del array no es una Promise, se trata como si fuera el valor de una Promise ya cumplida y simplemente se copia sin cambios en el array de salida.

La promesa devuelta por Promise.all() rechaza cuando cualquiera de las promesas de entrada es rechazada. Esto ocurre inmediatamente después del primer rechazo y puede ocurrir mientras otras Promesas de entrada siguen pendientes.

En ES2020, Promise.allSettled() toma una matriz de entrada Promises y devuelve una Promise, igual que hace Promise.all(). Pero Promise.allSettled() nunca rechaza la promesa devuelta, y no cumple esa promesa hasta que todas las promesas de entrada se hayan resuelto. La Promesa se resuelve en un array de objetos, con un objeto por cada Promesa de entrada. Cada uno de estos objetos devueltos tiene una propiedad de estado establecida como "cumplido" o "rechazado". Si el estado es "cumplido", entonces el objeto también tendrá una propiedad de valor que da el valor de cumplimiento. Y si el estado es "rechazado", entonces el objeto también tendrá una propiedad reason que da el valor de error o rechazo de la Promesa correspondiente:

```
Promise.allSettled([Promise.resolve(1), Promise.reject(2, 3)]).then(results => {
  results[0] //=> { status: "fulfilled", value: 1 } results[1] //=> { status: "rejected",
  reason: 2 } results[2] //=> { status: "cumplido", valor: 3 }
});
```

En ocasiones, es posible que quieras ejecutar varias promesas a la vez pero que sólo te importe el valor de la primera que se cumpla. En ese caso, puedes utilizar Promise.race() en lugar de Promise.all(). Es

devuelve una Promesa que se cumple o se rechaza cuando se cumple o se rechaza la primera de las Promesas del array de entrada. (O, si hay algún valor que no sea Promise en el array de entrada, simplemente devuelve el primero de ellos).

### 13.2.6 Hacer promesas

Hemos utilizado la función `fetch()`, que devuelve promesas, en muchos de los ejemplos anteriores porque es una de las funciones más simples incorporadas a los navegadores web que devuelven una promesa. Nuestra discusión sobre las promesas también se ha basado en las hipotéticas funciones `getJSON()` y `wait()` que devuelven promesas. Las funciones escritas para devolver promesas son realmente útiles, y esta sección muestra cómo puedes crear tus propias APIs basadas en promesas. En particular, mostraremos implementaciones de `getJSON()` y `wait()`.

## PROMESAS BASADAS EN OTRAS PROMESAS

Es fácil escribir una función que devuelva una Promise si tienes alguna otra función que devuelva una Promise para empezar. Dada una Promise, siempre puedes crear (y devolver) una nueva llamando a `.then()`. Así que si usamos la función `fetch()` existente como punto de partida, podemos escribir `getJSON()` así:

```
function getJSON(url) { return fetch(url).then(response => response.json()); }
```

El código es trivial porque el objeto `Response` de la API `fetch()` tiene un método `json()` predefinido. El método `json()` devuelve una Promise, que nosotros devolvemos desde nuestra devolución de llamada (la devolución de llamada es una función de flecha con un cuerpo de una sola expresión, por lo que la devolución es implícita),

por lo que la Promise devuelta por getJSON() se resuelve a la Promise devuelta por response.json(). Cuando esa Promise se cumple, la Promise devuelta por getJSON() se cumple con el mismo valor. Nótese que no hay manejo de errores en esta implementación de getJSON(). En lugar de comprobar response.ok y la cabecera Content-Type, simplemente permitimos que el método json() rechace la promesa devuelta con un SyntaxError si el cuerpo de la respuesta no puede ser analizado como JSON.

Escribamos otra función que devuelva una Promise, esta vez utilizando getJSON() como fuente de la Promise inicial:

```
function getHighScore() { return getJSON("/api/user/profile").then(profile =>
profile.highScore); }
```

Suponemos que esta función forma parte de algún tipo de juego basado en la web y que la URL "/api/user/profile" devuelve una estructura de datos con formato JSON que incluye una propiedad highScore.

## PROMESAS BASADAS EN VALORES SINCRÓNICOS

A veces, es posible que necesites implementar una API basada en Promise y devolver una Promise desde una función, aunque el cálculo a realizar no requiera realmente ninguna operación asíncrona. En ese caso, los métodos estáticos Promise.resolve() y Promise.reject() harán lo que quieras. Promise.resolve() toma un valor como único argumento y devuelve una Promise que se cumplirá inmediatamente (pero de forma asíncrona) con ese valor. Del mismo modo,

Promise.reject() toma un único argumento y devuelve una Promise que será rechazada con ese valor como razón. (Para que quede claro:

las Promesas devueltas por estos métodos estáticos no se cumplen o rechazan ya cuando se devuelven, sino que se cumplen o rechazan inmediatamente después de que el trozo de código síncrono actual haya terminado de ejecutarse. Normalmente, esto ocurre en unos pocos milisegundos, a menos que haya muchas tareas asíncronas pendientes esperando a ejecutarse).

Recordemos de §13.2.3 que una Promesa resuelta no es lo mismo que una Promesa cumplida. Cuando llamamos a `Promise.resolve()`, normalmente pasamos el valor de cumplimiento para crear un objeto Promise que muy pronto se cumplirá con ese valor. El método no se llama

`Promise.fulfill()`, sin embargo. Si se pasa una Promesa `p1` a `Promise.resolve()`, devolverá una nueva Promesa `p2`, que se resuelve inmediatamente, pero que no se cumplirá o rechazará hasta que se cumpla o rechace `p1`.

Es posible, aunque poco habitual, escribir una función basada en `Promise` en la que el valor se calcule de forma sincrónica y se devuelva de forma asíncrona con `Promise.resolve()`. Sin embargo, es bastante común tener casos especiales sincrónicos dentro de una función asíncrona, y puedes manejar estos casos especiales con `Promise.resolve()` y

`Promise.reject()`. En particular, si detecta condiciones de error (como valores de argumentos erróneos) antes de comenzar una operación asíncrona, puedes informar de ese error devolviendo una Promesa creada con `Promise.reject()`. (También podrías simplemente lanzar un error de forma sincrónica en ese caso, pero eso se considera mala forma porque entonces el que llama a tu función necesita escribir tanto una cláusula `catch` sincrónica como

usar un método `.catch()` asíncronico para manejar los errores).

Finalmente, `Promise.resolve()` es a veces útil para crear la Promesa inicial en una cadena de Promesas. Veremos un par de ejemplos que lo utilizan de esta manera.

## PROMESAS DESDE CERO

Tanto para `getJSON()` como para `getHighScore()`, empezamos por llamando a una función existente para obtener una Promesa inicial, y crear y devolver una nueva Promesa llamando al método `.then()` de esa Promesa inicial. ¿Pero qué pasa con la escritura de una función que devuelve una Promesa cuando no puedes usar otra función que devuelve una Promesa como punto de partida? En ese caso, se utiliza el constructor `Promise()` para crear un nuevo objeto Promise sobre el que se tiene un control total. Así es como funciona: invocas el constructor `Promise()` y le pasas una función como único argumento. La función que pases debe estar escrita para esperar dos parámetros, que, por convención, deben llamarse `resolve` y `reject`. El constructor llama sincrónicamente a tu función con argumentos de función para los parámetros `resolve` y `reject`. Después de llamar a tu función, el constructor de `Promise()` devuelve la Promesa recién creada. Esa Promesa devuelta está bajo el control de la función que has pasado al constructor. Esa función debe realizar alguna operación asíncrona y luego llamar a la función `resolve` para resolver o cumplir la Promesa devuelta o llamar a la función `reject` para rechazar la Promesa devuelta. Tu función no tiene por qué ser asíncrona: puede llamar a `resolve` o `reject` de forma síncrona, pero la Promesa se resolverá, cumplirá o rechazará de forma asíncrona si haces esto.

Puede ser difícil entender las funciones pasadas a un constructor con sólo leerlo, pero esperamos que algunos ejemplos lo aclaren. He aquí cómo escribir la función wait() basada en Promise que hemos utilizado en varios ejemplos anteriores del capítulo:

```
function wait(duration) { // Crea y devuelve una nueva Promise return new  
Promise((resolve, reject) => { // Estos controlan la Promise // Si el argumento no  
es válido, rechaza la Promise if (duration < 0) {  
    reject(new Error("El viaje en el tiempo aún no está implementado"));  
}  
// En caso contrario, espera de forma asíncrona y luego resuelve la Promesa.  
// setTimeout invocará a resolve() sin argumentos, lo que significa // que la  
Promise se cumplirá con el valor indefinido. setTimeout(resolve, duration);});  
}
```

Ten en cuenta que el par de funciones que utilizas para controlar el destino de una Promise creada con el constructor Promise() se llaman resolve() y reject(), no fulfill() y reject(). Si pasas una Promise a resolve(), la Promise devuelta se resolverá a esa nueva Promise. Sin embargo, a menudo se pasa un valor que no es una Promise, lo que hace que la Promise devuelta se cumpla con ese valor.

El ejemplo 13-1 es otro ejemplo de uso del constructor Promise(). Este implementa nuestra función getJSON() para su uso en Node, donde la API fetch() no está incorporada. Recuerda que comenzamos este capítulo con una discusión sobre callbacks y eventos asíncronos. Este ejemplo utiliza tanto callbacks como manejadores de eventos y es una buena demostración, por lo tanto, de cómo podemos

implementar APIs basadas en promesas sobre otros estilos de programación asíncrona.

### Ejemplo 13-1. Una función asíncrona getJSON()

---

```
const http = require("http");

functiongetJSON(url) { // Crea y devuelve una nueva Promise return new
Promise((resolve, reject) => { // Inicia una petición HTTP GET para la URL especificada
request = http.get(url, response => { // se llama cuando se inicia la respuesta // Rechaza
la Promise si el estado HTTP es incorrecto if (response.statusCode !== 200) { reject(new
Error(`HTTP status ${response.statusCode}`)); response.resume(); // para no perder
memoria
} // Y rechazar si las cabeceras de la respuesta son erróneas else if
(response.headers["content-type"] !== "application/json") { reject(new
Error("Invalid content-type")); response.resume(); // don't leak memory } else {
// En caso contrario, registra los eventos para leer el cuerpo de la respuesta let
body = ""; response.setEncoding("utf-8");

```

```

        response.on("data", chunk => { body += chunk; });
        response.on("end", () => {
            // Cuando el cuerpo de la respuesta esté completo, intenta analizarlo
            try { let
                parsed = JSON.parse(body);
                // Si se parsea con éxito, cumple la promesa resolve(parsed);
                } catch(e) {
                    // Si el análisis sintáctico falla, rechaza el
                    Promise.reject(e);
                }
            });
            // También rechazamos la Promesa si la petición falla antes de que
            // incluso obtener una respuesta (como cuando la red está caída) request.
            on("error", error => { reject(error); });
        });
    }
}

```

### 13.2.7 Promesas en secuencia

Promise.all() facilita la ejecución de un número arbitrario de promesas en paralelo. Y las cadenas de promesas facilitan la expresión de una secuencia de un número fijo de promesas. Sin embargo, ejecutar un número arbitrario de promesas en secuencia es más complicado. Supongamos, por ejemplo, que tienes una matriz de URLs para obtener, pero que para evitar sobrecargar tu red, quieres obtenerlas de una en una. Si la matriz tiene una longitud arbitraria y un contenido desconocido, no puedes escribir una cadena de promesas por adelantado, así que necesitas construir una dinámicamente, con código como este:

```

function fetchSequentially(urls) { // Aquí almacenaremos los cuerpos de las
  URLs a medida que los vayamos obteniendo const bodies = [];

    // Esta es una función que devuelve una promesa para obtener un cuerpo function
    fetchOne(url) { return fetch(url) . then(response => response.text())
      . then(cuerpo => {
        // Guardamos el cuerpo en el array, y a propósito
        // omitiendo un valor de retorno aquí (devolviendo undefined)
        bodies. push(body);
      });
    }

    // Comenzar con una promesa que se cumpla de inmediato
    // (con valor indefinido) let p = Promise.
    resolve(undefined);

    // Ahora, haga un bucle a través de las URLs deseadas, construyendo una cadena
    Promise
    // de longitud arbitraria, obteniendo una URL en cada etapa de la cadena for(url of
    urls) { p = p. then(() => fetchOne(url)); }

    // Cuando se cumple la última Promesa de esa cadena, entonces el
    // el array de cuerpos está listo. Así que vamos a devolver una Promesa para ese //
    array de cuerpos. Observa que no incluimos ningún manejador de errores:    //
    queremos permitir que los errores se propaguen a la persona que llama. return p.
    then(() => bodies);
  }
}

```

Con esta función `fetchSequentially()` definida, podríamos obtener las URLs de una en una con un código muy parecido al código `fetch-in-parallel` que usamos antes para demostrar `Promise.all()`:

```

fetchSequentially(urls) . then(bodies => { /* hacer algo con la matriz de
  cadenas */ }) . catch(e => console. error(e));

```

La función `fetchSequentially()` comienza creando una Promise que se cumplirá inmediatamente después de su devolución. Luego construye una larga cadena lineal de promesas a partir de esa promesa inicial y devuelve la última promesa de la cadena. Es como colocar una fila de fichas de dominó y luego derribar la primera.

Hay otro enfoque (posiblemente más elegante) que podemos adoptar. En lugar de crear las Promesas por adelantado, podemos hacer que el callback de cada Promesa cree y devuelva la siguiente Promesa. Es decir, en lugar de crear y encadenar un montón de promesas, creamos promesas que resuelven otras promesas. En lugar de crear una cadena de Promesas tipo dominó, estamos creando una secuencia de Promesas anidadas una dentro de otra como un conjunto de muñecas matrioskas. Con este enfoque, nuestro código puede devolver la primera promesa (la más externa), sabiendo que finalmente se cumplirá (o rechazará!) el mismo valor que la última

(más interna) de la secuencia. El

La función `promiseSequence()` que sigue está escrita para ser genérica y no es específica para la obtención de URLs. Está aquí al final de nuestra discusión sobre Promesas porque es complicada. Sin embargo, si has leído este capítulo con atención, espero que seas capaz de entender cómo funciona. En particular, observa que la función anidada dentro de `promiseSequence()` parece llamarse a sí misma de forma recursiva, pero como la llamada "recursiva" es a través de un método `then()`, en realidad no está ocurriendo ninguna recursión tradicional:

```
// Esta función toma un array de valores de entrada y una función "promiseMaker".
// Para cualquier valor de entrada x en el array, promiseMaker(x) debe devolver una
Promise
// que cumplirá con un valor de salida. Esta función devuelve un Promise
// que se llena a un array de los valores de salida calculados.
//
// En lugar de crear las promesas de una vez y dejar que se ejecuten en
// En paralelo, sin embargo, promiseSequence() sólo ejecuta una Promise a la vez
// y no llama a promiseMaker() para un valor hasta que la promesa anterior // se
haya cumplido. function promiseSequence(inputs, promiseMaker) { // Hacer una
copia privada del array que podamos modificar inputs = [... inputs];

    // Esta es la función que usaremos como callback de Promise // Esta es la magia
    pseudorecursiva que hace que todo esto funcione.

    function handleNextInput(outputs) { if (inputs. length
== 0) {
        // Si no quedan más entradas, entonces devuelve el array
        // de salidas, cumpliendo finalmente esta Promesa y todas las // Promesas
anteriores resueltas-pero-no-cumplidas. return outputs;
    } si no {
        // Si todavía hay valores de entrada para procesar, entonces
        // devuelve un objeto Promise, resolviendo el
Promesa
        // con el valor futuro de una nueva Promesa.
        let nextInput = inputs. shift(); // Obtener el siguiente valor de entrada,
return promiseMaker(nextInput) // calcula el siguiente valor de salida,
        // A continuación, crea una nueva matriz de salidas con el nuevo valor de
salida . then(output => outputs. concat(output))
        // Luego "recurre", pasando el nuevo array más largo de salidas .
then(handleNextInput);
    }
}
```

```
// Empezar con una Promesa que se cumpla a un array vacío y usar // la función anterior como su callback.  
return Promise.resolve([]).then(handleNextInput); }
```

Esta función promiseSequence() es intencionalmente genérica.

Podemos usarla para obtener URLs con código como este:

```
// Dada una URL, devuelve una Promesa que cumple con el texto del cuerpo de la URL  
function fetchBody(url) { return fetch(url).then(r =>  
r.text()); } // Usarlo para obtener secuencialmente un montón de cuerpos de URL  
promiseSequence(urls, fetchBody) .then(bodies => /* hacer algo con el array de cadenas */) .catch(console.error);
```

## 13.3 async y await

ES2017 introduce dos nuevas palabras clave -async y await- que representan un cambio de paradigma en la programación asíncrona de JavaScript. Estas nuevas palabras clave simplifican drásticamente el uso de las promesas y nos permiten escribir código asíncrono basado en promesas que parece código síncrono que se bloquea mientras espera respuestas de la red u otros eventos asíncronos. Aunque sigue siendo importante entender cómo funcionan las promesas, gran parte de su complejidad (y a veces incluso su propia presencia!) se desvanece al utilizarlas con async y await.

Como hemos comentado anteriormente en el capítulo, el código asíncrono no puede devolver un valor o lanzar una excepción de la misma forma que el código síncrono normal. Y es por esto que las Promesas están diseñadas de la manera en que lo están. El valor de una Promise cumplida es como el valor de retorno de una función sincrónica. Y el valor de una Promise rechazada es como el valor lanzado por una función sincrónica. Esta última similitud se hace

explícita por el nombre del método `.catch()`. `async` y `await` toman el código eficiente basado en `Promise` y ocultan las `Promises` para que tu código asíncrono pueda ser tan fácil de leer y tan fácil de razonar como el código ineficiente, bloqueante y síncrono.

### 13.3.1 esperar expresiones

La palabra clave `await` toma una `Promise` y la convierte en un valor de retorno o en una excepción lanzada. Dado un objeto `Promise` `p`, la expresión `await p` espera hasta que `p` se cumpla. Si `p` se cumple, entonces el valor de `await p` es el valor de cumplimiento de `p`. Por otro lado, si `p` es rechazado, entonces la expresión `await p` lanza el valor de rechazo de `p`. Normalmente no usamos `await` con una variable que contiene una `Promise`; en su lugar, la usamos antes de la invocación de una función que devuelve una `Promise`:

```
let response = await fetch("/api/user/profile"); let profile = await  
response.json();
```

Es fundamental entender de inmediato que la palabra clave `await` no hace que tu programa se bloquee y literalmente no haga nada hasta que la promesa especificada se asiente. El código sigue siendo asíncrono, y el `await` simplemente disfraza este hecho. Esto significa que *cualquier código que utilice await es en sí mismo asíncrono*.

### 13.3.2 Funciones asíncronas

Dado que cualquier código que utiliza `await` es asíncrono, hay una regla crítica: *sólo puedes utilizar la palabra clave await dentro de funciones que han sido declaradas con la palabra clave async*. Aquí está una versión de la función `getHighScore()` de antes en el capítulo, reescrita para usar `async` y `await`:

```
async function getHighScore() { let response = await  
fetch("/api/user/profile"); let profile = await response.json(); return  
profile.highScore; }
```

Declarar una función `async` significa que el valor de retorno de la función será una `Promise` aunque no aparezca código relacionado con `Promise` en el cuerpo de la función. Si una función asíncrona parece devolver normalmente, entonces el objeto `Promise` que es el verdadero valor de retorno de la función se resolverá a ese valor de retorno aparente. Y si una función asíncrona parece lanzar una excepción, entonces el objeto `Promise` que devuelve será rechazado con esa excepción.

La función `getHighScore()` está declarada asíncrona, por lo que devuelve una `Promise`. Y como devuelve una `Promise`, podemos usar la palabra clave `await` con ella:

```
displayHighScore(await getHighScore());
```

Pero recuerda que esa línea de código sólo funcionará si está dentro de otra función asíncrona. Puedes anidar expresiones `await` dentro de funciones `async` tan profundamente como quieras. Pero si estás en el nivel superior<sup>2</sup> o estás dentro de una función que no es asíncrona por alguna razón, entonces no puedes usar `await` y tienes que tratar con una `Promise` devuelta de la manera habitual:

```
getHighScore().then(displayHighScore).catch(console.error);
```

Puedes utilizar la palabra clave `async` con cualquier tipo de función. Funciona con la palabra clave `function` como declaración o como expresión. Funciona con funciones en forma de flecha y con la forma abreviada de método en clases y literales de objeto.

(Consulta el [capítulo 8](#) para saber más sobre las distintas formas de escribir funciones).

### 13.3.3 Esperando múltiples promesas

Supongamos que hemos escrito nuestra función getJSON() usando `async`:

```
async function getJSON(url) { let response = await
fetch(url); let body = await response.json(); return
body; }
```

Y ahora supongamos que queremos obtener dos valores JSON con esta función:

```
let valor1 = await getJSON(url1);
```

```
let value2 = await getJSON(url2);
```

El problema de este código es que es innecesariamente secuencial: la obtención de la segunda URL no comenzará hasta que la primera se haya completado. Si la segunda URL no depende del valor obtenido de la primera URL, entonces probablemente deberíamos intentar obtener los dos valores al mismo tiempo. Este es un caso en el que se muestra la naturaleza basada en promesas de las funciones asíncronas. Para esperar un conjunto de funciones asíncronas que se ejecutan de forma concurrente, utilizamos `Promise.all()` tal y como haríamos si trabajáramos con Promises directamente:

```
let [valor1, valor2] = await Promise.all([getJSON(url1), getJSON(url2)]);
```

### 13.3.4 Detalles de la implementación

Por último, para entender cómo funcionan las funciones asíncronas, puede ser útil pensar en lo que ocurre bajo el capó.

Supongamos que escribes una función asíncrona como esta

```
async function f(x) { /* body */ }
```

Puedes pensar en esto como una función de devolución de promesas envuelta en el cuerpo de tu función original:

```
function f(x) { return new Promise(function(resolve, reject) { try {
  resolve((function(x) { /* body */ })(x));
} catch(e) { reject(e); }
}); }
```

Es más difícil expresar la palabra clave `await` en términos de una transformación sintáctica como ésta. Pero piensa en la palabra clave `await` como un marcador que divide el cuerpo de una función en trozos separados y sincrónicos. Un intérprete de ES2017 puede dividir el cuerpo de la función en una secuencia de subfunciones separadas, cada una de las cuales se pasa al método `then()` de la promesa marcada con `await` que la precede.

## 13.4 Iteración asíncrona

Comenzamos este capítulo con una discusión sobre la asincronía basada en llamadas y eventos, y cuando introdujimos las Promesas, señalamos que eran útiles para cálculos asíncronos de una sola vez, pero que no eran adecuadas para su uso con fuentes de eventos asíncronos repetitivos, como `setInterval()`, el evento "click" en un navegador web, o el evento "data" en un flujo Node. Debido a que

las Promesas simples no funcionan para secuencias de eventos asíncronos, tampoco podemos usar funciones asíncronas regulares y las sentencias await para estas cosas.

Sin embargo, ES2018 ofrece una solución. Los iteradores asíncronos son como los iteradores descritos en el [capítulo 12](#), pero están basados en promesas y están pensados para ser utilizados con una nueva forma del bucle for/of: for/await.

### 13.4.1 El bucle for/await

El Nodo 12 hace que sus flujos legibles sean asíncronamente iterables. Esto significa que puedes leer trozos sucesivos de datos de un flujo con un bucle for/await como este:

```
const fs = require("fs");

async function parseFile(filename) { let stream = fs.createReadStream(filename, {
encoding: "utf-8"}); for await (let chunk of stream) { parseChunk(chunk); // Asume
que parseChunk() está definido en otro lugar }
}
```

Al igual que una expresión await normal, el bucle for/await está basado en promesas. A grandes rasgos, el iterador asíncrono produce una promesa y el bucle for/await espera a que esa promesa se cumpla, asigna el valor de cumplimiento a la variable del bucle y ejecuta el cuerpo del bucle. Y luego vuelve a empezar, obteniendo otra Promise del iterador y esperando a que esa nueva Promise se cumpla.

Suponga que tiene una matriz de URLs:

```
const urls = [url1, url2, url3];
```

Puedes llamar a `fetch()` en cada URL para obtener un array de Promesas:

```
const promises = urls.map(url => fetch(url));
```

Ya vimos anteriormente en el capítulo que podíamos utilizar `Promise.all()` para esperar a que se cumplan todas las promesas del array. Pero supongamos que queremos los resultados de la primera búsqueda tan pronto como estén disponibles y no queremos esperar a que se obtengan todas las URLs. (Por supuesto, la primera búsqueda puede tardar más que las demás, por lo que esto no es necesariamente más rápido que usar `Promise.all()`). Los arrays son iterables, así que podemos iterar a través del array de promesas con un bucle `for/of` normal:

```
for(const promise of promises) { response =  
    await promise; handle(response); }
```

Este código de ejemplo utiliza un bucle `for/of` normal con un iterador normal. Pero como este iterador devuelve Promesas, también podemos utilizar el nuevo `for/await` para un código ligeramente más sencillo:

```
for await (const response of promises) {  
    handle(response); }
```

En este caso, el bucle `for/await` sólo construye la llamada `await` en el bucle y hace nuestro código ligeramente más compacto, pero los dos ejemplos hacen exactamente lo mismo. Es importante destacar que ambos ejemplos sólo funcionarán si están dentro de funciones declaradas `async`; un bucle `for/await` no es diferente de una expresión `await` normal en ese sentido.

Es importante darse cuenta, sin embargo, de que estamos usando `for/await` con un iterador regular en este ejemplo. Las cosas son más interesantes con iteradores totalmente asíncronos.

### 13.4.2 Iteradores asíncronos

Repasemos algo de la terminología del [capítulo 12](#). Un objeto *iterable* es aquel que puede ser utilizado con un bucle `for/of`. Define un método con el nombre simbólico `Symbol.iterator`. Este método devuelve un objeto *iterador*. El objeto iterador tiene un método `next()`, que puede ser llamado repetidamente para obtener los valores del objeto iterable. El método `next()` del objeto iterador devuelve objetos resultado de *la iteración*. El objeto resultado de la iteración tiene una propiedad `value` y/o una propiedad `done`.

Los iteradores asíncronos son bastante similares a los iteradores normales, pero hay dos diferencias importantes. En primer lugar, un objeto iterable asíncrono implementa un método con el nombre simbólico

`Symbol.asyncIterator` en lugar de `Symbol.iterator`. (Como vimos antes, `for/await` es compatible con los objetos iterables normales pero prefiere los objetos iterables asíncronos, e intenta el `Symbol.asyncIterator` antes de probar el método `Symbol.iterator`). En segundo lugar, el método `next()` de un iterador asíncrono devuelve una `Promise` que resuelve un objeto resultado del iterador en lugar de devolver un objeto resultado del iterador directamente.

### NOTA

En la sección anterior, cuando usamos `for/await` en un array de Promesas regular, iterable sincrónicamente, estábamos trabajando con objetos resultado de iteradores sincrónicos en los que la propiedad `value` era un objeto Promise pero la propiedad `done` era sincrónica. Los iteradores asíncronos verdaderos devuelven Promesas para los objetos resultado de la iteración, y tanto la propiedad `value` como la `done` son asíncronas. La diferencia es sutil: con los iteradores asíncronos, la elección sobre cuándo termina la iteración puede hacerse de forma asíncrona.

### 13.4.3 Generadores asíncronos

Como vimos en el [capítulo 12](#), la forma más sencilla de implementar un iterador suele ser utilizar un generador. Lo mismo ocurre con los iteradores asíncronos, que podemos implementar con funciones generadoras que declaramos asíncronas. Un generador asíncrono tiene las características de las funciones asíncronas y las características de los generadores: puedes usar `await` como lo harías en una función asíncrona normal, y puedes usar `yield` como lo harías en un generador normal. Pero los valores que se ceden se envuelven automáticamente en promesas. Incluso la sintaxis de los generadores asíncronos es una combinación: la función asíncrona y la función `*` se combinan en la función asíncrona `*`. Aquí hay un ejemplo que muestra cómo se puede utilizar un generador asíncrono y un bucle `for/await` para ejecutar repetidamente el código a intervalos fijos utilizando la sintaxis de bucle en lugar de una función de devolución de llamada `setInterval()`:

```
// Una envoltura basada en Promise alrededor de setTimeout() con la que podemos
// usar await.
// Devuelve una Promise que se cumple en el número de milisegundos especificado
function elapsedTime(ms) { return new Promise(resolve => setTimeout(resolve, ms));
}
```

```
// Una función generadora asíncrona que incrementa un contador y lo devuelve // un
número especificado (o infinito) de veces en un intervalo especificado.
async function* clock(interval, max=Infinity) { for(let count = 1; count <= max;
count++) { // regular for loop await elapsedTime(interval); // esperar a que pase el
tiempo yield count; // yield the counter }
}

// Una función de prueba que utiliza el generador asíncrono con for/await async
function test() { // Asíncrono para que podamos utilizar for/await for await (let tick of
clock(300, 100)) { // Bucle 100 veces cada 300ms console.log(tick); }
}
```

### 13.4.4. Implementación de Iteradores Asíncronos

En lugar de utilizar generadores asíncronos para implementar iteradores asíncronos, también es posible implementarlos directamente definiendo un objeto con un método `Symbol.asyncIterator()` que devuelva un objeto con un método `next()` que devuelva una `Promise` que resuelva a un objeto resultado del iterador. En el siguiente código, reimplementamos la función `clock()` del ejemplo anterior para que no sea un generador y en su lugar sólo devuelva un objeto iterable asíncrono. Observa que el método `next()` en este ejemplo no devuelve explícitamente una `Promise`; en su lugar, simplemente declaramos que `next()` es asíncrono:

```

función reloj(intervalo, max=Infinito) {
    // Una versión modificada por Promise de setTimeout con la que podemos usar await.
    // Tenga en cuenta que esto toma un tiempo absoluto en lugar de un intervalo.
    function until(time) { return new Promise(resolve => setTimeout(resolve, time -
Date. now())); }

    // Devuelve un objeto iterable asíncrono return { startTime: Date. now(), //
Recuerda cuándo empezamos count: 1, // Recuerda en qué iteración estamos

        async next() { // El método next() lo convierte en un iterador if (this. count >
max) { // ¿Hemos terminado?
            return { done: true }; // Resultado de la iteración indicando done }
            // Averigua cuándo debe comenzar la siguiente iteración, deja que targetTime
= this. startTime + this. count * interval; // espera hasta ese momento, await
until(targetTime);
            // y devolver el valor de la cuenta en un objeto resultado de la iteración.
            return { valor: este. count++ };
        },
        // Este método significa que este objeto iterador es también un iterable.
        [Symbol.asyncIterator]() { devuelve esto; }
    };
}

```

Esta versión de la función clock() basada en el iterador corrige un fallo de la versión basada en el generador. Observe que, en este código más nuevo, apuntamos al tiempo absoluto en el que debe comenzar cada iteración y restamos el tiempo actual de eso para calcular el intervalo que pasamos a setTimeout(). Si usamos clock() con un bucle for/await, esta versión ejecutará las iteraciones del

bucle con mayor precisión en el intervalo especificado porque tiene en cuenta el tiempo necesario para ejecutar el cuerpo del bucle. Pero este arreglo no es sólo acerca de la exactitud del tiempo. El bucle for/await siempre espera a que se cumpla la promesa devuelta por una iteración antes de comenzar la siguiente. Pero si usas un iterador asíncrono sin un bucle for/await, no hay nada que te impida llamar al método next() cuando quieras. Con la versión basada en el generador de clock(), si se llama al método

next() tres veces de forma secuencial, obtendrás tres Promesas que se cumplirán casi exactamente al mismo tiempo, lo que probablemente no es lo que quieras. La versión basada en el iterador que hemos implementado aquí no tiene ese problema.

La ventaja de los iteradores asíncronos es que nos permiten representar flujos de eventos o datos asíncronos. La función clock() comentada anteriormente era bastante sencilla de escribir porque la fuente de la asincronía eran las llamadas a setTimeout() que hacíamos nosotros mismos. Pero cuando tratamos de trabajar con otras fuentes asíncronas, como la activación de los manejadores de eventos, se vuelve sustancialmente más difícil implementar iteradores asíncronos: normalmente tenemos una sola función manejadora de eventos que responde a los eventos, pero cada llamada al método next() del iterador debe devolver un objeto Promise distinto, y pueden ocurrir múltiples llamadas a next() antes de que se resuelva la primera Promise. Esto significa que cualquier método iterador asíncrono debe ser capaz de mantener una cola interna de Promesas que resuelve en orden a medida que se producen los eventos asíncronos. Si encapsulamos este comportamiento de cola de promesas en una clase AsyncQueue, será

mucho más fácil escribir iteradores asíncronos basados en AsyncQueue.<sup>3</sup>

La clase AsyncQueue que sigue tiene los métodos enqueue() y dequeue() como es de esperar para una clase de cola. Sin embargo, el método dequeue() devuelve una Promise en lugar de un valor real, lo que significa que está bien llamar a dequeue() antes de que enqueue() haya sido llamado. La clase AsyncQueue es también un iterador asíncrono, y está pensada para ser utilizada con un bucle for/await cuyo cuerpo se ejecuta una vez cada vez que se pone en cola un nuevo valor de forma asíncrona. (AsyncQueue tiene un método close()). Una vez llamado, no se pueden poner en cola más valores. Cuando una cola cerrada está vacía, el bucle for/await dejará de hacer un bucle).

Ten en cuenta que la implementación de AsyncQueue no utiliza async ni await y en su lugar trabaja directamente con Promises. El código es algo complicado, y puedes utilizarlo para comprobar tu comprensión del material que hemos cubierto en este largo capítulo. Aunque no entiendas del todo la implementación de AsyncQueue, echa un vistazo al ejemplo más corto que le sigue: implementa un simple pero muy interesante iterador asíncrono sobre AsyncQueue.

```
/**  
 * Una clase de cola iterable asíncrona. Añade valores con enqueue()  
 * y eliminarlos con dequeue(). dequeue() devuelve un  
 Promesa, que  
 * significa que los valores pueden ser retirados de la cola antes de ser puestos en  
 la cola. La página web  
 * implementa [Symbol.asyncIterator] y next() para que pueda  
 * con el bucle for/await (que no terminará hasta que  
 * se llama al método close()).  
 */ class AsyncQueue {  
 constructor() {  
 // Los valores que se han puesto en cola pero no se han retirado todavía se  
 // almacenan aquí this.values = [];  
 // Cuando las Promesas son retiradas de la cola antes de que sus valores  
 // correspondientes sean // colocados en la cola, los métodos de resolución para esas  
 // Promesas son almacenados aquí.  
 this.resolvers = [];  
 // Una vez cerrado, no se pueden poner en cola más valores, y no se devuelven  
 // más promesas // no cumplidas. this.closed = false;
```

```
}

enqueue(value) { if (this. closed) { throw new Error("AsyncQueue
closed"); } if (this. resolvers. length > 0) {
    // Si este valor ya ha sido prometido, resuelve esa Promise const resolve
= this. resolvers. shift(); resolve(value); } else { // En caso contrario, ponlo en
cola this. values. push(value); } }

dequeue() { if (this. values. length > 0) {
    // Si hay un valor en cola, devuelve un resuelto
Promise para ello const value = this. values. shift(); return
Promise. resolve(value); } else if (this. closed) {
    // Si no hay valores en cola y estamos cerrados, devuelve una promesa // resuelta
para el marcador de "fin de flujo" return Promise. resolve(AsyncQueue. EOS); } else {
    // En caso contrario, devuelve una Promesa no resuelta,
    // poner en cola la función resolver para su uso posterior return new
Promise((resolver) => { this. resolvers. push(resolver); }); } }

close() {
    // Una vez que se cierra la cola, no se encolan más valores.
```

```

// Así que resuelve cualquier promesa pendiente con el marcador de fin de flujo
while(this.resolvers.length > 0) { this.resolvers.shift()(AsyncQueue.EOF); } this.
closed = true;

// Define el método que hace que esta clase sea iterable de forma asíncrona [Symbol.
asyncliterator]() { return this; }

// Definir el método que hace de este un iterador asíncrono. El
// dequeue() La promesa resuelve un valor o el centinela EOS si estamos
// cerrado. Aquí, necesitamos devolver una Promise que resuelva un
objeto // iterador de resultados. next() {

    return this.dequeue().then(value => (value === AsyncQueue.EOF) ? { valor:
indefinido, hecho: verdadero } : { valor: valor, hecho: falso }}); }
}

// Un valor centinela devuelto por dequeue() para marcar el "fin del flujo" cuando se
cierra
AsyncQueue.EOF = Symbol("end-of-stream");

```

Dado que esta clase AsyncQueue define los fundamentos de la iteración asíncrona, podemos crear nuestros propios e interesantes iteradores asíncronos simplemente poniendo en cola valores de forma asíncrona. Aquí hay un ejemplo que utiliza AsyncQueue para producir un flujo de eventos del navegador web que puede ser manejado con un bucle for/await:

```

// Empujar eventos del tipo especificado en el
Elemento del documento

```

```
// en un objeto AsyncQueue, y devuelve la cola para su uso como flujo de eventos
function eventStream(elt, type) { const q = new AsyncQueue(); // Crea una cola elt.
addEventListener(type, e=> q.enqueue(e)); // Encarga los eventos return q; }

async function handleKeys() {
  // Obtener un flujo de eventos de pulsación de teclas y hacer un bucle para cada uno
  for await (const event of eventStream(document, "keypress")) { console.log(event.key);
}
}
```

## 13.5 Resumen

En este capítulo has aprendido:

- La mayor parte de la programación de JavaScript en el mundo real es asíncrona.

Tradicionalmente, la asíncronía se ha manejado con eventos y funciones de callback. Sin embargo, esto puede ser complicado, porque puedes terminar con múltiples niveles de callbacks anidados dentro de otros callbacks, y porque es difícil hacer un manejo robusto de los errores.

- Las promesas proporcionan una nueva forma de estructurar las funciones de devolución de llamada. Si se utilizan correctamente (y, por desgracia, las promesas son fáciles de utilizar de forma incorrecta), pueden convertir el código asíncrono que se habría anidado en cadenas lineales de llamadas a `then()` en las que un paso asíncrono de un cálculo sigue a otro. Además, las promesas permiten centralizar la gestión de errores

en una sola llamada a `catch()` al final de una cadena de llamadas a `then()`.

- Las palabras clave `async` y `await` nos permiten escribir código asíncrono que está basado en Promise bajo el capó pero que parece código síncrono. Esto hace que el código sea

más fácil de entender y razonar. Si una función se declara asíncrona, devolverá implícitamente una promesa. Dentro de una función asíncrona, puedes esperar una Promise (o una función que devuelva una Promise) como si el valor de la Promise se calculara de forma sincrónica.

- Los objetos que son asíncronamente iterables pueden ser utilizados con un bucle for/await. Puedes crear objetos asíncronamente iterables implementando un método [Symbol.asyncIterator] () o invocando una función asíncrona \* generadora. Los iteradores asíncronos proporcionan una alternativa a los eventos de "datos" en los flujos en Node y pueden ser utilizados para representar un flujo de eventos de entrada del usuario en el lado del cliente de JavaScript.

---

La clase XMLHttpRequest no tiene nada en particular que ver con XML. En los clientes modernos

<sup>1</sup> de JavaScript, ha sido sustituida en gran medida por la API fetch(), que se trata en [§15.11.1](#). El ejemplo de código mostrado aquí es el último ejemplo basado en XMLHttpRequest que queda en este libro.

Normalmente se puede utilizar await en el nivel superior de la consola de desarrollo de un navegador. Y

<sup>2</sup> hay una propuesta pendiente para permitir el nivel superior de espera en una futura versión de JavaScript.

Conocí este enfoque de la iteración asíncrona en el blog del Dr. Axel <sup>3</sup> Rauschmayer, <https://2ality.com>.



# Capítulo 14.

# Metaprogramación

---

Este capítulo cubre una serie de características avanzadas de JavaScript que no se utilizan comúnmente en la programación diaria, pero que pueden ser valiosas para los programadores que escriben bibliotecas reutilizables y de interés para cualquier persona que quiera juguetear con los detalles sobre cómo se comportan los objetos de JavaScript.

Muchas de las características descritas aquí pueden describirse vagamente como "metaprogramación": si la programación normal es escribir código para manipular datos, la metaprogramación es escribir código para manipular otro código. En un lenguaje dinámico como JavaScript, los límites entre la programación y la metaprogramación son borrosos: incluso la simple capacidad de iterar sobre las propiedades de un objeto con un bucle `for/in` podría ser considerada "meta" por los programadores acostumbrados a lenguajes más estáticos.

Los temas de metaprogramación que se tratan en este capítulo son:

- §14.1 Controlar la enumeración, la eliminación y la configuración de las propiedades de los objetos
- §14.2 Controlar la extensibilidad de los objetos y crear

objetos "sellados" y "congelados"

- §14.3 Consulta y fijación de los prototipos de los objetos
- §14.4 Ajuste del comportamiento de sus tipos con los conocidos Símbolos
- §14.5 Creación de DSLs (lenguajes específicos de dominio) con funciones de etiquetas de plantillas
  - §14.6 Exploración de objetos con métodos reflectantes
  - §14.7 Controlar el comportamiento de los objetos con Proxy

## 14.1 Atributos de las propiedades

Las propiedades de un objeto de JavaScript tienen nombres y valores, por supuesto, pero cada propiedad también tiene tres atributos asociados que especifican cómo se comporta esa propiedad y qué se puede hacer con ella:

- El atributo *writable* especifica si el valor de una propiedad puede cambiar o no.
- El atributo *enumerable* especifica si la propiedad es enumerada por el bucle `for/in` y el método `Object.keys()`.
- El atributo *configurable* especifica si una propiedad puede ser eliminada y también si los atributos de la propiedad pueden ser modificados.

Las propiedades definidas en los literales de los objetos o por asignación ordinaria a un objeto son escribibles, enumerables y configurables. Pero muchas de las propiedades definidas por la biblioteca estándar de JavaScript no lo son.

Esta sección explica la API para consultar y establecer los atributos de las propiedades. Esta API es especialmente importante para los autores de bibliotecas porque:

- Les permite añadir métodos a los objetos prototipo y hacerlos enumerables, como los métodos incorporados.
- Les permite "bloquear" sus objetos, definiendo propiedades que no pueden ser modificadas ni eliminadas.

Recordemos que en §6.10.6, mientras que las "propiedades de datos" tienen un valor, las "propiedades accesorias" tienen un método getter y/o setter. Para los propósitos de esta sección, vamos a considerar que los métodos getter y setter de una propiedad accesor son atributos de la propiedad. Siguiendo esta lógica, incluso diremos que el valor de una propiedad de datos es también un atributo. Así, podemos decir que una propiedad tiene un nombre y cuatro atributos. Los cuatro atributos de una propiedad de datos son *valor*, *escribible*, *enumerable* y *configurable*. Las propiedades accesorias no tienen un atributo de *valor* o un atributo de escritura: su escritura está determinada por la presencia o ausencia de un setter. Así que los cuatro atributos de una propiedad accesoria son *get*, *set*, *enumerable* y *configurable*.

Los métodos de JavaScript para consultar y establecer los atributos de una propiedad utilizan un objeto llamado descriptor de *propiedad* para representar el conjunto de cuatro atributos. Un objeto descriptor de propiedades tiene propiedades con los mismos nombres que los atributos de la propiedad que describe. Así, el objeto descriptor de propiedades de una propiedad de datos tiene propiedades denominadas

valor, escribible, enumerable y configurable. Y el para una propiedad accesoria tiene propiedades get y set en lugar de value y writable. Las propiedades writable, enumerable y configurable son valores booleanos, y las propiedades get y set son valores de función.

Para obtener el descriptor de una propiedad con nombre de un objeto especificado, llame a Object.getOwnPropertyDescriptor():

```
// Devuelve {valor: 1, escribible:true, enumerable:true, configurable:true}
Objeto.getOwnPropertyDescriptor({x: 1}, "x");

// Aquí hay un objeto con una propiedad de acceso de sólo lectura const random = {
get octeto() { return Math.floor(Math.random()*256); }, };

// Devuelve { get: /*func*/, set:undefined, enumerable:true, configurable:true}
Objeto.getOwnPropertyDescriptor(random, "octeto");

// Devuelve undefined para propiedades heredadas y propiedades que no existen.
Object.getOwnPropertyDescriptor({}, "x") //=> undefined; no hay tal prop Object.
getOwnPropertyDescriptor({}, "toString") //=> undefined; heredado
```

Como su nombre indica, Object.getOwnPropertyDescriptor() sólo funciona para las propiedades propias. Para consultar los atributos de las propiedades heredadas, debe recorrer explícitamente la cadena de prototipos. (Véase Object.getPrototypeOf() en §14.3); véase también el Reflect.getOwnPropertyDescriptor() en §14.6.)

Para establecer los atributos de una propiedad o para crear una nueva propiedad con los atributos especificados, llame a

`Object.defineProperty()`, pasando el objeto a modificar, el nombre de la propiedad a crear o alterar y el objeto descriptor de la propiedad:

```
let o = {};// Empieza sin ninguna propiedad // Añade una propiedad de datos
no enumerable x con valor 1. Object. defineProperty(o, "x", { valor: 1,
escribible: true, enumerable: false,
configurable: true });

// Comprueba que la propiedad está ahí pero no es numerable
o.x //=> 1 Object. keys(o) //=> []

// Ahora modifica la propiedad x para que sea de sólo lectura
Object. defineProperty(o, "x", { writable: false });

// Intenta cambiar el valor de la propiedad
o.x = 2; // Falla silenciosamente o lanza TypeError en modo estricto
o.x //=> 1

// La propiedad sigue siendo configurable, por lo que podemos cambiar su valor así
Object. defineProperty(o, "x", { valor: 2 });

o.x //=> 2

// Ahora cambia x de una propiedad de datos a una propiedad accesoria Object.
defineProperty(o, "x", { get: function() { return 0; } });

o.x //=> 0
```

El descriptor de la propiedad que se pasa a `Object.defineProperty()` no tiene que incluir los cuatro atributos. Si está creando una nueva propiedad, los atributos omitidos se consideran falsos o indefinidos. Si estás modificando una propiedad existente, los atributos que omitas simplemente se dejarán sin modificar. Tenga en cuenta que este método altera una propiedad propia existente o crea una nueva propiedad propia, pero no alterará una propiedad heredada. Véase también la función muy similar

`Reflect.defineProperty()` en §14.6.

Si desea crear o modificar más de una propiedad a la vez, utilice `Object.defineProperties()`. El primer argumento es el objeto que se va a modificar. El segundo argumento es un objeto que asigna los nombres de las propiedades que se van a crear o modificar a los descriptores de propiedades de dichas propiedades. Por ejemplo:

```
let p = Object. definePropiedades({}, { x: { valor: 1, escribible: true, enumerable: true, configurable: true }, y: { valor: 1, escribible: true, enumerable: true, configurable: true }, r: { get() { return Math. sqrt(this. x*this. x + this. y*this. y); }, enumerable: true, configurable: true } });
p. r // => Math.SQRT2
```

Este código comienza con un objeto vacío, y luego le añade dos propiedades de datos y una propiedad accesoria de sólo lectura. Se basa en el hecho de que `Object.defineProperties()` devuelve el objeto modificado (al igual que `Object.defineProperty()`).

El método `Object.create()` fue introducido en [§6.2](#). Allí aprendimos que el primer argumento de ese método es el objeto prototipo del objeto recién creado. Este método también acepta un segundo argumento opcional, que es el mismo que el segundo argumento de `Object.defineProperties()`. Si pasas un conjunto de descriptores de propiedades a `Object.create()`, entonces se utilizan para añadir propiedades al objeto recién creado.

`Object.defineProperty()` y `Object.defineProperties()` lanzan `TypeError` si el intento de crear o modificar una propiedad no está permitido. Esto ocurre si se intenta añadir una nueva propiedad a un objeto no extensible (ver [§14.2](#)). Las otras razones por las que estos métodos pueden lanzar `TypeError` tienen que ver con los propios atributos. El atributo *writable* gobierna los intentos de cambiar el atributo *value*. Y el atributo *configurable* gobierna los

*intentos de* cambiar los otros atributos (y también especifica si una propiedad puede ser eliminada). Sin embargo, las reglas no son del todo sencillas. Es posible cambiar el valor de una propiedad no escribible si esa propiedad es configurable, por ejemplo. También es posible cambiar una propiedad de escribible a no escribible incluso si esa propiedad no es configurable. Aquí están las reglas completas. Las llamadas a

`Object.defineProperty()` o

`Object.defineProperties()` que intentan violarlas lanzan un `TypeError`:

- Si un objeto no es extensible, puede editar sus propiedades propias existentes, pero no puede añadirle nuevas propiedades.
- Si una propiedad no es configurable, no se pueden cambiar sus atributos configurables o enumerables.
- Si una propiedad accesoria no es configurable, no puedes cambiar su método getter o setter, y no puedes cambiarla a una propiedad de datos.
- Si una propiedad de datos no es configurable, no se puede cambiar a una propiedad accesoria.
- Si una propiedad de datos no es configurable, no se puede cambiar su atributo *writable* de false a true, pero sí de true a false.
- Si una propiedad de datos no es configurable ni escribible, no se puede cambiar su valor. Sin embargo, puedes cambiar el valor de una propiedad que es configurable pero no escribible (porque eso sería lo mismo que hacerla escribible, luego cambiar el valor, y luego volver a convertirla en no escribible).

En §6.7 se ha descrito la función `Object.assign()` que copia los valores de las propiedades de uno o más objetos de origen en un objeto de destino.

`Object.assign()` sólo copia las propiedades enumerables, y los valores de las propiedades, no los atributos de las mismas. Esto es normalmente lo que queremos, pero significa, por ejemplo, que si uno de los objetos fuente tiene una propiedad accesoria, es el valor devuelto por la función getter el que se copia al objeto destino, no la propia función getter. [El ejemplo 14-1](#) demuestra cómo podemos utilizar

`Object.getOwnPropertyDescriptor()` y `Object.defineProperty()` para crear una variante de `Object.assign()` que copia descriptores de propiedades enteros en lugar de sólo copiar valores de propiedades.

#### *Ejemplo 14-1. Copiar propiedades y sus atributos de un objeto a otro*

---

```
/*
 *      Definir una nueva función Object.assignDescriptors() que funcione como
 *      Object.assign() excepto que copia los descriptores de propiedades de
 *      objetos de origen en el objeto de destino en lugar de limitarse a copiar
 *      valores de las propiedades. Esta función copia todas las propiedades propias,
 *      tanto
 *      enumerables y no enumerables. Y porque copia los descriptores,
 *      copia las funciones getter de los objetos fuente y sobrescribe las setter
 *      en el objeto de destino en lugar de invocar esos getters y
```

```
*   los colocadores.

*
*   Object.assignDescriptors() propaga cualquier TypeErrors lanzado por
*   Object.defineProperty(). Esto puede ocurrir si el objeto de destino está sellado
*   o congelado o si alguna de las propiedades de la fuente intenta cambiar una
*   propiedad no configurable en el objeto de destino.
*
*   Tenga en cuenta que la propiedad assignDescriptors se añade a Object con
*   Object.defineProperty() para que la nueva función pueda ser creada como
*   una propiedad no enumerable como Object.assign().
*/
Object. defineProperty(Object, "assignDescriptors", { // Coincide con los
atributos de Object.assign() escribible: true, enumerable: false,
configurable: true,
    // La función que es el valor de la propiedad assignDescriptors.
    value: function(target, ... sources) { for(let source of sources) { for(let name of Object.
getOwnPropertyNames(source))
{
    let desc = Object. getOwnPropertyDescriptor(source, name);
    Object. defineProperty(target, name, desc); }

    for(let símbolo de
Object. getOwnPropertySymbols(source)) { let desc = Object.
getOwnPropertyDescriptor(source, symbol);
        Objeto. defineProperty(target, symbol, desc);
    } } return target; }
});
```

```
let o = {c: 1, get count() {return this.c++;}}; // Definir el objeto con getter let p =  
Object.assign({}, o); // Copiar los valores de las propiedades let q = Object.  
assignDescriptors({}, o); // Copiar los descriptores de las propiedades  
p.count //=> 1: Esto es ahora sólo una propiedad de datos por lo que  
p.count //=> 1: ...el contador no se incrementa.  
q.count //=> 2: Se incrementó una vez cuando lo copiamos la primera vez,  
q.count //=> 3: ...pero hemos copiado el método getter para que se incremente.
```

## 14.2 Extensibilidad de los objetos

El atributo *extensible* de un objeto especifica si se pueden añadir nuevas propiedades al objeto o no. Los objetos ordinarios de JavaScript son extensibles por defecto, pero puedes cambiarlo con las funciones descritas en esta sección.

Para determinar si un objeto es extensible, pásalo a `Object.isExtensible()`. Para hacer que un objeto no sea extensible, páselo a `Object.preventExtensions()`. Una vez hecho esto, cualquier intento de añadir una nueva propiedad al objeto lanzará un `TypeError` en modo estricto y simplemente fallará silenciosamente sin un error en modo no estricto. Además, intentar cambiar el prototipo (ver §14.3) de un objeto no extensible siempre lanzará un `TypeError`.

Tenga en cuenta que no hay manera de hacer que un objeto sea extensible de nuevo una vez que lo ha hecho no extensible.

También hay que tener en cuenta que llamar a `Object.preventExtensions()` sólo afecta a la extensibilidad del propio objeto. Si se añaden nuevas propiedades al prototipo de un objeto no extensible, el objeto no extensible heredará esas nuevas propiedades.

Dos funciones similares, `Reflect.isExtensible()` y `Reflect.preventExtensions()`, se describen en [§14.6](#).

El propósito del atributo *extensible* es poder "bloquear" los objetos en un estado conocido y evitar la manipulación externa. El atributo *extensible* de los objetos se utiliza a menudo junto con los atributos *configurable* y *escribable* de las propiedades, y JavaScript define funciones que facilitan el establecimiento de estos atributos juntos:

- `Object.seal()` funciona como `Object.preventExtensions()`, pero además de Al hacer que el objeto no sea extensible, también hace que todas las propiedades propias de ese objeto no sean configurables. Esto significa que no se pueden añadir nuevas propiedades al objeto, y que las existentes no se pueden borrar ni configurar. Sin embargo, las propiedades existentes que son escribibles pueden seguir siendo configuradas. No hay forma de desprecintar un objeto sellado. Puede utilizar `Object.isSealed()` para determinar si un objeto está sellado.
- `Object.freeze()` bloquea los objetos de forma aún más estricta. Además de hacer que el objeto no sea extensible y sus propiedades no sean configurables, también hace que todas las propiedades de datos propias del objeto sean de sólo lectura. (Si el objeto tiene propiedades accesorias con métodos setter, éstos no se ven afectados y pueden seguir siendo invocados por asignación a la propiedad). Utilice `Object.isFrozen()` para determinar si un objeto está congelado.

Es importante entender que `Object.seal()` y `Object.freeze()` sólo afectan al objeto que se les pasa: no tienen efecto sobre el prototipo de ese objeto. Si quieres bloquear a fondo un objeto, probablemente necesites sellar o congelar también los objetos de la cadena de prototipos.

`Object.preventExtensions()`, `Object.seal()`, y `Object.freeze()` devuelven el objeto que se les pasa, lo que significa que se pueden utilizar en invocaciones de funciones anidadas:

```
// Crea un objeto sellado con un prototipo congelado y una propiedad no numerable let  
o = Object.seal(Object.create(Object.freeze({x: 1}), {y: {valor: 2, escribible: true}}));
```

Si está escribiendo una biblioteca de JavaScript que pasa objetos a funciones de devolución de llamada escritas por los usuarios de su biblioteca, puede utilizar `Object.freeze()` en esos objetos para evitar que el código del usuario los modifique. Esto es fácil y conveniente de hacer, pero hay ventajas: los objetos congelados pueden interferir con las estrategias comunes de prueba de JavaScript, por ejemplo.

## 14.3 El atributo prototipo

El atributo prototipo de un objeto especifica el objeto del que hereda propiedades. (Revise §6.2.3 y §6.3.2 para más información sobre los prototipos y la herencia de propiedades.) Este es un atributo tan importante que normalmente decimos simplemente "el prototipo de o" en lugar de "el atributo prototipo de o".

Recuerde también que cuando `prototype` aparece en la fuente de código, se refiere a una propiedad ordinaria del objeto, no al atributo `prototype`: En [el capítulo 9](#) se explicó que la propiedad `prototype` de una función constructora especifica el atributo `prototype` de los objetos creados con ese constructor.

El atributo `prototype` se establece cuando se crea un objeto. Los objetos creados a partir de literales de objeto utilizan `Object.prototype` como su prototipo. Los objetos creados con `new` utilizan como prototipo el valor de la propiedad `prototype` de su

función constructora. Y los objetos creados con `Object.create()` utilizan el primer argumento de esa función (que puede ser `null`) como su prototipo.

Puede consultar el prototipo de cualquier objeto pasando ese objeto a

`Object.getPrototypeOf():`

```
Object.getPrototypeOf({}) //=> Object.prototype  
Object.getPrototypeOf([]) //=> Array.prototype  
Object.getPrototypeOf(()=>{}) //=> Function.prototype
```

Una función muy similar, `Reflect.getPrototypeOf()`, se describe en [§14.6](#).

Para determinar si un objeto es el prototipo de (o forma parte de la cadena de prototipos de) otro objeto, utilice el método `isPrototypeOf()`:

```
let p = {x: 1}; // Definir un objeto prototipo.  
let o = Object.create(p); // Crear un objeto con ese prototipo.  
p.isPrototypeOf(o) //=> true: o hereda de p  
Object.prototype.isPrototypeOf(p) //=> true: p hereda de  
Object.prototype  
Object.prototype.isPrototypeOf(o) //=> true: o también lo hace
```

Tenga en cuenta que `isPrototypeOf()` realiza una función similar a la del operador `instanceof` (véase [§4.9.4](#)).

El atributo prototipo de un objeto se establece cuando se crea el objeto y normalmente permanece fijo. Sin embargo, puedes cambiar el prototipo de un objeto con `Object.setPrototypeOf()`:

```
let o = {x: 1}; let p = {y: 2}; Object.setPrototypeOf(o, p); // Establecer el prototipo de o a p  
o.y // => 2: ahora o hereda la propiedad y let a = [1, 2, 3];  
Object.setPrototypeOf(a, p); // Establecer el prototipo de la matriz a en p  
a.join // => undefined: a ya no tiene un método join()
```

Por lo general, no es necesario utilizar nunca `Object.setPrototypeOf()`. Las implementaciones de JavaScript pueden hacer optimizaciones agresivas basadas en la suposición de que el prototipo de un objeto es fijo e inmutable. Esto significa que si alguna vez se llama a `Object.setPrototypeOf()`, cualquier código que utilice los objetos alterados puede ejecutarse mucho más lentamente de lo que lo haría normalmente.

Una función similar, `Reflect.setPrototypeOf()`, se describe en [§14.6](#).

Algunas de las primeras implementaciones de JavaScript en los navegadores exponían el atributo de prototipo de un objeto a través de la propiedad `__proto__`.

(escrito con dos guiones bajos al principio y al final). Hace mucho tiempo que está en desuso, pero hay suficiente código en la web que depende de `__proto__` como para que el estándar ECMAScript lo exija para todas las implementaciones de JavaScript que se ejecuten en los navegadores web. (Node también lo soporta, aunque el estándar no lo requiere para Node).

JavaScript, `__proto__` es legible y escribible, y puede (aunque no debería) utilizarlo como alternativa a `Object.getPrototypeOf()` y `Object.setPrototypeOf()`. Un uso interesante de

`__proto__`, sin embargo, es para definir el prototipo de un literal de objeto:

```
let p = {z: 3}; let o = { x:  
 1, y: 2, __proto__: p };  
o.z // => 3: o hereda de p
```

## 14.4 Símbolos conocidos

El tipo `Symbol` se añadió a JavaScript en ES6, y una de las principales razones para hacerlo fue añadir de forma segura extensiones al lenguaje sin romper la compatibilidad con el código ya desplegado en la web. Vimos un ejemplo de esto en [el Capítulo 12](#), donde aprendimos que puedes hacer que una clase sea iterable implementando un método cuyo "nombre" es el `Symbol Symbol.iterator`.

`Symbol.iterator` es el ejemplo más conocido del "conocido Símbolos". Se trata de un conjunto de valores de `Symbol` almacenados como propiedades de la función de fábrica `Symbol()` que se utilizan para permitir que el código JavaScript controle ciertos comportamientos de bajo nivel de objetos y clases. Las subsecciones que siguen describen cada uno de estos conocidos `Symbols` y explican cómo pueden utilizarse.

### 14.4.1 `Symbol.iterator` y `Symbol.asyncIterator`

Los símbolos `Symbol.iterator` y `Symbol.asyncIterator` permiten a los objetos o clases hacerse iterables o asíncronamente iterables. Fueron cubiertos en detalle en [el Capítulo 12](#) y [§13.4.2](#), respectivamente, y se mencionan de nuevo aquí sólo para completar.

## 14.4.2 Symbol.hasInstance

Cuando se describió el operador `instanceof` en §4.9.4, dijimos que el lado derecho debe ser una función constructora y que la expresión `o instanceof f` se evaluaba buscando el valor `f.prototype` dentro de la cadena de prototipos de `o`. Esto sigue siendo cierto, pero en ES6 y en adelante, `Symbol.hasInstance` proporciona una alternativa. En ES6, si el lado derecho de `instanceof` es cualquier objeto con un método `[Symbol.hasInstance]`, entonces ese método es invocado con el valor del lado izquierdo como argumento, y el valor de retorno del método, convertido en booleano, se convierte en el valor del operador `instanceof`. Y, por supuesto, si el valor del lado derecho no tiene un método `[Symbol.hasInstance]` sino que es una función, entonces el operador `instanceof` se comporta de forma ordinaria.

`Symbol.hasInstance` significa que podemos utilizar la función `instanceof` para realizar una comprobación de tipos genérica con objetos de pseudotipo convenientemente definidos. Por ejemplo:

```
// Definir un objeto como un "tipo" que podemos utilizar con instanceof
let uint8 = {
  [Symbol.hasInstance](x) { return Number.isInteger(x) && x >= 0 && x <= 255; }
};

128 instanceof uint8 // => true
256 instanceof uint8 // => false: demasiado grande
Matemáticas. PI instanceof uint8 // => false: no es un entero
```

Tenga en cuenta que este ejemplo es inteligente pero confuso porque utiliza un objeto no-clase donde normalmente se esperaría una clase. Sería igual de fácil -y más claro para los lectores de su código- escribir una función `isUInt8()` en lugar de confiar en este comportamiento `Symbol.hasInstance`.

### 14.4.3 Símbolo.toStringTag

Si se invoca el método `toString()` de un objeto básico de JavaScript, se obtiene la cadena "[objeto Objeto]":

```
{}.toString() //=> "[objeto Objeto]"
```

Si se invoca esta misma función `Object.prototype.toString()` como método de instancias de tipos incorporados, se obtienen algunos resultados interesantes:

```
Object.prototype.toString.call([]) //=> "[object Array]"
Objeto.prototype.toString.call(/./) //=> "[objeto
RegExp]"
Object.prototype.toString.call(()=>{}) //=> "[objeto
Función]"
Object.prototype.toString.call("") //=> "[objeto
Cadena]"
Object.prototype.toString.call(0) //=> "[objeto
```

```
Número]"
Object.prototype.toString.call(false) //=> "[objeto
Boolean]"
```

Resulta que se puede utilizar este  
La técnica `Object.prototype.toString().call()` con cualquier  
de JavaScript para obtener el "atributo de clase" de un objeto que  
contiene información de tipo que no está disponible de otra manera.  
La siguiente función `classof()` es posiblemente más útil que el  
operador `typeof`, que no distingue entre tipos de objetos:

```

function classof(o) { return Object.prototype.toString.call(o).slice(8,-1); }

classof(null) //=> "Null" classof(undefined) //=>
"Undefined" classof(1) //=> "Number" classof(10n**100n)
//=> "BigInt" classof("") //=> "String" classof(false) //=>
"Boolean" classof(Symbol()) //=> "Symbol" classof({}) //=>
"Object" classof([]) //=> "Array" classof(/./) //=> "RegExp"
classof(()=>{}) //=> "Function" classof(new Map()) //=>
"Map" classof(new Set()) //=> "Set" classof(new Date()) //=>
= "Fecha"

```

Antes de ES6, este comportamiento especial del método `Object.prototype.toString()` sólo estaba disponible para instancias de tipos incorporados, y si se llama a esta función `classof()` en una instancia de una clase definida por uno mismo, simplemente devolverá "Object". En ES6, sin embargo, `Object.prototype.toString()` busca una propiedad con el nombre simbólico `Symbol.toStringTag` en su argumento, y si tal propiedad existe, utiliza el valor de la propiedad en su salida. Esto significa que si defines una clase propia, puedes hacerla funcionar fácilmente con funciones como `classof()`:

```

class Rango { get [Symbol.toStringTag]() { return "Rango"; } // el resto
de esta clase se omite aquí }
let r = new Range(1, 10); Object.prototype.toString.call(r) //=> "[objeto Range]"
classof(r) //=> "Range"

```

#### 14.4.4 Símbolo.especie

Antes de ES6, JavaScript no proporcionaba ninguna forma real de crear subclases robustas de clases incorporadas como Array. En ES6, sin embargo, puedes extender cualquier clase incorporada simplemente usando las palabras clave `class` y `extends`. En §9.5.2 se demostró esto con esta simple subclase de Array:

```
// Una subclase trivial de Array que añade getters para el primer y el último elemento.
class EZArray extends Array { get first() { return this[0]; } get last()
{ return this[this.length-1]; } }

let e = new EZArray(1,2,3); let f = e.map(x
=> x * x);
e.last // => 3: el último elemento de EZArray e
f.last // => 9: f es también un EZArray con una propiedad last
```

Array define los métodos `concat()`, `filter()`, `map()`, `slice()` y `splice()`, que devuelven arrays. Cuando creamos una subclase de array como EZArray que hereda estos métodos, ¿debe el método heredado devolver instancias de Array o instancias de EZArray? Se pueden dar buenos argumentos para cualquiera de las dos opciones, pero la especificación ES6 dice que (por defecto) los cinco métodos que devuelven arrays devolverán instancias de la subclase.

Así es como funciona:

- En ES6 y posteriores, el constructor `Array()` tiene una propiedad con el nombre simbólico `Symbol.species`. ( Nótese que este `Symbol` se utiliza como nombre de una propiedad de la función del constructor. La mayoría de los otros Símbolos conocidos descritos aquí se utilizan como el nombre de los métodos de un objeto prototípico).

- Cuando creamos una subclase con extends, el constructor resultante de la subclase hereda propiedades del constructor de la superclase. (Esto es adicional al tipo de herencia normal, donde las instancias de la subclase heredan métodos de la superclase). Esto significa que el constructor de cada subclase de Array también tiene una propiedad heredada con el nombre Symbol.species. (O una subclase puede definir su propia propiedad con este nombre, si lo desea).
- Los métodos como map() y slice() que crean y devuelven nuevas matrices se han modificado ligeramente en ES6 y posteriores. En lugar de crear un Array normal, invocan (en efecto) el nuevo this.constructor[Symbol.species]() para crear el nuevo array.

Ahora viene lo interesante. Supongamos que Array[Symbol.species] era una propiedad de datos normal, definida así:

```
Array[Symbol.species] = Array;
```

En ese caso, los constructores de las subclases heredarían el constructor Array() como su "especie", e invocar map() en una subclase de array devolvería una instancia de la superclase en lugar de una instancia de la subclase. Sin embargo, no es así como se comporta ES6. La razón es que Array[Symbol.species] es una propiedad de sólo lectura cuya función getter simplemente devuelve esto. Los constructores de subclases heredan esta función getter, lo que significa que por defecto, cada constructor de subclase es su propia "especie".

Sin embargo, a veces este comportamiento por defecto no es lo que quieres. Si quieres que los métodos de EZArray devuelvan objetos Array normales, sólo tienes que establecer EZArray[Symbol.species]

como Array. Pero como la propiedad heredada es un accesorio de sólo lectura, no puedes simplemente establecerla con un operador de asignación. Sin embargo, puedes utilizar `defineProperty()`:

```
EZArray[Symbol.species] = Array; // El intento de establecer una propiedad de sólo lectura falla
```

```
// En su lugar podemos utilizar defineProperty():
Object.defineProperty(EZArray, Symbol.species, {value:
Array});
```

La opción más sencilla es probablemente definir explícitamente su propio getter `Symbol.species` al crear la subclase en primer lugar:

```
class EZArray extends Array { static get [Symbol.species]() { return Array; }
get first() { return this[0]; }

get last() { return this[this.length-1]; } }

let e = new EZArray(1,2,3); let f = e.map(x
=> x - 1);
e.último // => 3
f.last // => undefined: f es un array normal sin last getter
```

La creación de subclases útiles de Array fue el principal caso de uso que motivó la introducción de `Symbol.species`, pero no es el único lugar donde se utiliza este conocido `Symbol`. Las clases de matrices tipificadas utilizan el `Symbol` de la misma manera que lo hace la clase Array. Del mismo modo, el método `slice()` de `ArrayBuffer` mira el

`Symbol.species` de este constructor en lugar de simplemente crear un nuevo `ArrayBuffer`. Y los métodos `Promise` como `then()` que devuelven nuevos objetos `Promise` también crean esos objetos a través de este protocolo `species`. Por último, si te encuentras subclasiificando `Map` (por ejemplo) y definiendo métodos que

devuelven nuevos objetos Map, es posible que quieras utilizar Symbol.species tú mismo en beneficio de las subclases de tu subclase.

#### 14.4.5 Symbol.isConcatSpreadable

El método Array concat() es uno de los métodos descritos en la sección anterior que utiliza Symbol.species para determinar qué constructor utilizar para el array devuelto. Pero concat() también utiliza Symbol.isConcatSpreadable. Recuerde de §7.8.3 que el método concat() de un array trata su valor this y sus argumentos de array de forma diferente a sus argumentos que no son de array: los argumentos que no son de array simplemente se añaden al nuevo array, pero el array this y cualquier argumento de array se aplana o "extienden" de forma que los elementos del array se concatenan en lugar del propio argumento del array.

Antes de ES6, concat() se limitaba a utilizar Array.isArray() para determinar si debía tratar un valor como un array o no. En ES6, el algoritmo ha cambiado ligeramente: si el argumento (o el valor) de concat() es un objeto y tiene una propiedad con el nombre simbólico Symbol.isConcatSpreadable, entonces el valor booleano de esa propiedad se utiliza para determinar si el argumento debe ser "extendido". Si no existe tal propiedad, entonces se utiliza Array.isArray() como en versiones anteriores del lenguaje.

Hay dos casos en los que podría querer utilizar este símbolo:

- Si crea un objeto tipo Array (véase §7.9) y quiere que se comporte como un array real cuando se le pasa a concat(), puede simplemente añadir la propiedad simbólica a su objeto:

```

let arraylike = { longitud:
  1, 0: 1,
  [Símbolo. isConcatSpreadable]: true
};
[].concat(arraylike) // => [1]: (sería [[1]] si no se
extendiera)

```

- Las subclases de arrays son extensibles por defecto, así que si estás definiendo una subclase de array que no quieras que actúe como un array cuando se usa con concat(), entonces puedes1 añadir un getter como este a tu subclase:

```

class NonSpreadableArray extends Array { get [Symbol.
isConcatSpreadable]() { return false; } } let a = new
NonSpreadableArray(1,2,3);
[].concat(a).length // => 1; (tendría 3 elementos si a estuviera
extendido)

```

#### 14.4.6 Símbolos de coincidencia de patrones

En §11.3.2 se documentan los métodos String que realizan operaciones de concordancia de patrones utilizando un argumento RegExp. En ES6 y posteriores, estos métodos se han generalizado para trabajar con objetos RegExp o cualquier objeto que defina el comportamiento de la concordancia de patrones mediante propiedades con nombres simbólicos. Para cada uno de los métodos de cadena match(), matchAll(), search(), replace(), y split(), existe el correspondiente Symbol conocido: Symbol.match, Symbol.search, etc.

Los RegExps son una forma general y muy potente de describir patrones textuales, pero pueden ser complicados y no se adaptan bien a las coincidencias difusas. Con los métodos de cadena generalizados, puede definir sus propias clases de patrones utilizando los conocidos métodos `Symbol` para proporcionar una comparación personalizada. Por ejemplo, podría realizar comparaciones de cadenas utilizando `Intl.Collator` (véase §11.7.3) para ignorar los acentos al realizar la comparación. O puede definir una clase de patrón basada en el algoritmo `Soundex` para comparar palabras basándose en sus sonidos aproximados o para comparar cadenas hasta una determinada distancia Levenshtein.

En general, cuando se invoca uno de estos cinco métodos `String` en un objeto patrón como éste:

```
string.method(pattern, arg)
```

esa invocación se convierte en una invocación de un método con nombre simbólico en su objeto patrón:

```
patrón[símbolo](cadena, arg)
```

Como ejemplo, considere la clase de concordancia de patrones del siguiente ejemplo, que implementa la concordancia de patrones utilizando los simples comodines `*` y `?` que probablemente conozca de los sistemas de archivos. Este estilo de concordancia de patrones se remonta a los primeros días del sistema operativo Unix, y los patrones a menudo se llaman *globos*:

```

class Glob { constructor(glob) { this.glob
= glob;

    // Implementamos la coincidencia global usando RegExp internamente.
    // ? coincide con cualquier carácter excepto /, y * coincide con cero o más
// de esos caracteres. Utilizamos grupos de captura alrededor de cada uno.
    let regexpText = glob.replace("?", " ([^/])").replace("*", "([^/]*)");

    // Utilizamos la bandera u para obtener una coincidencia compatible con Unicode.
    // Los globos están pensados para coincidir con cadenas enteras, por lo que
utilizamos el ^ y el $
    // anclas y no implementar search() o matchAll() ya que // no son
útiles con patrones como este.
    this.regexp = new RegExp(`^${regexpText}$`, "u"); } toString() { return this.
glob; }

```

```

[Symbol. search](s) { return s. search(this.regexp); }
[Symbol. match](s) { return s. match(this.regexp); } [Symbol. replace](s,
replacement) { return s. replace(this.regexp, replacement); } }

let pattern = new Glob("docs/*.txt");
"docs/js.txt".search(pattern) // => 0: coincide con el carácter
0 "docs/js.htm".search(pattern) // => -1: no coincide let match = "docs/js.txt".
match(pattern); match[0] // => "docs/js.txt" match[1] //=> "js" match.index //
=> 0
"docs/js.txt".replace(pattern, "web/$1.htm") // =>
"web/js.htm"

```

## 14.4.7 Símbolo.toPrimitive

En §3.9.3 se explica que JavaScript tiene tres algoritmos ligeramente diferentes para convertir objetos en valores primitivos. En términos

generales, para las conversiones en las que se espera o se prefiere un valor de cadena, JavaScript invoca primero el método `toString()` de un objeto y recurre al método `valueOf()` si `toString()` no está definido o no devuelve un valor primitivo. Para las conversiones en las que se prefiere un valor numérico, JavaScript intenta primero el método `valueOf()` y recurre a `toString()` si `valueOf()` no está definido o si no devuelve un valor primitivo. Y, por último, en los casos en que no hay preferencia, deja que la clase decida cómo hacer la conversión. Los objetos de fecha se convierten utilizando primero `toString()`, y todos los demás tipos intentan primero `valueOf()`.

En ES6, el conocido Símbolo `Symbol.toPrimitive` le permite anular este comportamiento predeterminado de objeto a primitivo y le da un control completo sobre cómo las instancias de sus propias clases se convertirán en valores primitivos. Para ello, defina un método con este nombre simbólico. El método debe devolver un valor primitivo que represente de alguna manera el objeto. El método que defina será invocado con un único argumento de cadena que le indica qué tipo de conversión está intentando hacer JavaScript en su objeto:

- Si el argumento es "string", significa que JavaScript está haciendo la conversión en un contexto en el que esperaría o preferiría (pero no requiere) una cadena. Esto sucede cuando se interpola el objeto en un literal de plantilla, por ejemplo.
- Si el argumento es "number", significa que JavaScript está haciendo la conversión en un contexto en el que esperaría o preferiría (pero no requiere) un valor numérico. Esto ocurre cuando se utiliza el objeto con un operador `<` o `>` o con operadores aritméticos como `-` y `*`.

- Si el argumento es "por defecto", significa que JavaScript está convirtiendo su objeto en un contexto en el que podría funcionar un valor numérico o de cadena. Esto ocurre con los operadores +, == y !=.

Muchas clases pueden ignorar el argumento y simplemente devolver el mismo valor primitivo en todos los casos. Si quieres que las instancias de tu clase sean comparables y ordenables con < y >, entonces esa es una buena razón para definir un método [Symbol.toPrimitive].

#### 14.4.8 Símbolo.inescopiable

El último Símbolo bien conocido que cubriremos aquí es uno oscuro que fue introducido como una solución a los problemas de compatibilidad causados por la declaración obsoleta with.

Recordemos que la sentencia with toma un objeto y ejecuta el cuerpo de la sentencia como si estuviera en un ámbito donde las propiedades de ese objeto fueran variables. Esto causaba problemas de compatibilidad cuando se añadían nuevos métodos a la clase Array, y rompía parte del código existente. Symbol.unscopables es el resultado. En ES6 y posteriores, la sentencia with se ha modificado ligeramente. Cuando se utiliza con un objeto o, una sentencia with calcula

Object.keys(o[Symbol.unscopables] || {}) e ignora propiedades cuyos nombres están en el array resultante al crear el ámbito simulado en el que ejecutar su cuerpo. ES6 utiliza esto para añadir nuevos métodos a Array.prototype sin romper el código existente en la web. Esto significa que puede encontrar una lista de los métodos Array más nuevos evaluando:

```
let newArrayMethods = Object.keys(Array.prototype[Symbol.unscopables]);
```

## 14.5 Etiquetas de las plantillas

Las cadenas dentro de los signos de puntuación se conocen como "literales de plantilla" y se trataron en §3.3.4. Cuando una expresión cuyo valor es una función va seguida de un literal de plantilla, se convierte en una invocación a una función, y la llamamos "literal de plantilla etiquetado". La definición de una nueva función de etiqueta para su uso con literales de plantilla etiquetados puede considerarse como porque las plantillas etiquetadas se utilizan a menudo para definir DSLs (lenguajes específicos de dominio) y definir una nueva función de etiqueta es como añadir una nueva sintaxis a JavaScript. Los literales de las plantillas etiquetadas han sido adoptados por una serie de paquetes de JavaScript para el frontend. El lenguaje de consulta GraphQL utiliza una función de etiqueta gql`` para permitir que las consultas se incrusten en el código JavaScript. Y la biblioteca Emotion utiliza una función de etiqueta css`` para permitir que los estilos CSS se incrusten en JavaScript. Esta sección muestra cómo escribir tus propias funciones de etiqueta como éstas.

Las funciones de las etiquetas no tienen nada de especial: son ordinarias

Las funciones de JavaScript, y no se requiere ninguna sintaxis especial para definirlas. Cuando una expresión de función va seguida de un literal de plantilla, se invoca la función. El primer argumento es un array de cadenas, y a éste le siguen cero o más argumentos adicionales, que pueden tener valores de cualquier tipo.

El número de argumentos depende del número de valores que se interpolen en el literal de la plantilla. Si el literal de la plantilla es simplemente una cadena constante sin interpolaciones, entonces la función de etiqueta será llamada con una matriz de esa cadena y sin argumentos adicionales. Si el literal de la plantilla incluye un valor interpolado, entonces se llama a la función de etiqueta con dos argumentos. El primero es una matriz de dos cadenas, y el segundo es el valor interpolado. Las cadenas de esa matriz inicial son la cadena a la izquierda del valor interpolado y la cadena a su derecha, y cualquiera de ellas puede ser la cadena vacía. Si el literal de la plantilla incluye dos valores interpolados, la función de etiqueta se invoca con tres argumentos: una matriz de tres cadenas y los dos valores interpolados. Las tres cadenas (cualquiera o todas pueden estar vacías) son el texto a la izquierda del primer valor, el texto entre los dos valores y el texto a la derecha del segundo valor. En el caso general, si el literal de la plantilla tiene  $n$  valores interpolados, se invocará la función de etiqueta con  $n+1$  argumentos. El primer argumento será una matriz de  $n+1$  cadenas, y los argumentos restantes son los  $n$  valores interpolados, en el orden en que aparecen en el literal de la plantilla.

El valor de un literal de plantilla es siempre una cadena. Pero el valor de un literal de plantilla etiquetado es cualquier valor que la función de etiqueta devuelva. Esto puede ser una cadena, pero cuando la función de etiqueta se utiliza para implementar un DSL, el valor de retorno es típicamente una estructura de datos que no es una cadena y que es una representación analizada de la cadena.

Como ejemplo de una función de etiqueta de plantilla que devuelve una cadena, considere la siguiente plantilla html``, que es útil cuando

quiere interpolar valores de forma segura en una cadena de HTML. La etiqueta realiza el escape de HTML en cada uno de los valores antes de utilizarlo para construir la cadena final:

```
function html(cadenas, ... valores) {
    // Convierte cada valor en una cadena y escapa los caracteres especiales de HTML let
    escaped = values.map(v => String(v) . replace("&", "&amp;"))
        . replace("<", "&lt;")
        . replace(">", "&gt;")
        . replace('"', "&quot;")
        . replace("'", "&#39;"));

    // Devuelve las cadenas concatenadas y los valores escapados let result =
    cadenas[0]; for(let i = 0; i < escaped.length; i++) { result += escaped[i] +
    cadenas[i+1]; } return result;

let operator = "<"; html`<b>x ${operator} y</b>` // => "<b>x &lt;
y</b>"
```

```
let kind = "game", name = "D&D"; html`<div class="${kind}"> ${name}</div>` //
=>'<div class="game">D&amp;D</div>'
```

Para un ejemplo de una función de etiqueta que no devuelve una cadena sino una representación analizada de una cadena, piense en la clase patrón Glob definida en §14.4.6. Dado que el constructor Glob() toma un único argumento de cadena, podemos definir una función de etiqueta para crear nuevos objetos Glob:

```

function glob(strings, ... values) { // Ensamblar las cadenas y los valores en una sola
  cadena let s = strings[0]; for(let i = 0; i < values.length; i++) { s += values[i] +
  strings[i+1];
  } // Devuelve una representación analizada de esa cadena return new
  Glob(s); }

let root = "/tmp"; let filePattern = glob`${root}/*.html`; // Una alternativa
RegExp
"/tmp/test.html".match(filePattern)[1] //=> "test"

```

Una de las características mencionadas de pasada en el [apartado 3.3.4](#) es la

Función de etiqueta String.raw`` que devuelve una cadena en su forma "cruda" sin interpretar ninguna de las secuencias de escape de la barra invertida. Esto se implementa utilizando una característica de la invocación de la función etiqueta que no hemos discutido todavía. Cuando se invoca una función de etiqueta, hemos visto que su primer argumento es un array de cadenas. Pero este array también tiene una propiedad llamada raw, y el valor de esa propiedad es otro array de cadenas, con el mismo número de elementos. La matriz de argumentos incluye cadenas a las que se les han interpretado las secuencias de escape como es habitual. Y el array raw incluye cadenas en las que no se interpretan las secuencias de escape. Esta característica oscura es importante si quieras definir un DSL con una gramática que utilice barras invertidas. Por ejemplo, si quisieramos que nuestra función de la etiqueta glob`` soportara la coincidencia de patrones en rutas al estilo de Windows (que utilizan barras invertidas en lugar de barras inclinadas) y no quisieramos que los usuarios de la etiqueta tuvieran que duplicar cada barra invertida, podríamos reescribir esa función

para que utilizara `strings.raw[]` en lugar de `strings[]`. La desventaja, por supuesto, sería que ya no podríamos usar escapes como \u en nuestros literales glob.

## 14.6 La API de Reflect

El objeto Reflect no es una clase; al igual que el objeto Math, sus propiedades simplemente definen una colección de funciones relacionadas. Estas funciones, añadidas en ES6, definen una API para "reflexionar sobre" objetos y sus propiedades. Hay poca funcionalidad nueva aquí: el objeto Reflect define un conveniente conjunto de funciones, todas en un único espacio de nombres, que imitan el comportamiento de la sintaxis del lenguaje principal y duplican las características de varias funciones Object preexistentes.

Aunque las funciones Reflect no proporcionan ninguna característica nueva, agrupan las características en una API conveniente. Y, lo que es más importante, el conjunto de funciones Reflect se corresponde con el conjunto de métodos de manejo de Proxy que conoceremos en [§14.7](#).

La API de Reflect consta de las siguientes funciones:

### *Reflect.apply(f, o, args)*

Esta función invoca la función f como un método de o (o la invoca como una función sin este valor si o es nulo) y pasa los valores de la matriz args como argumentos. Equivale a `f.apply(o, args)`.

### *Reflect.construct(c, args, newTarget)*

Esta función invoca el constructor c como si se hubiera utilizado la palabra clave new y pasa los elementos del array args como argumentos. Si se especifica el argumento opcional newTarget, se utiliza como valor de new.target dentro de la invocación del constructor. Si no se especifica, el valor de new.target será c.

#### *Reflect.defineProperty(o, name, descriptor)*

Esta función define una propiedad en el objeto o, utilizando name (una cadena o símbolo) como nombre de la propiedad. El objeto Descriptor debe definir el valor (o getter y/o setter) y los atributos de la propiedad. Reflect.defineProperty() es muy similar a Object.defineProperty() pero devuelve true en caso de éxito y false en caso de fallo. (Object.defineProperty() devuelve o en caso de éxito y lanza TypeError en caso de fallo).

#### *Reflect.deleteProperty(o, name)*

Esta función borra la propiedad con el nombre simbólico o de cadena especificado del objeto o, devolviendo true si tiene éxito (o si no existe tal propiedad) y false si la propiedad no pudo ser borrada. Llamar a esta función es similar a escribir delete o[nombre].

#### *Reflect.get(o, nombre, receptor)*

Esta función devuelve el valor de la propiedad de o con el nombre especificado (una cadena o símbolo). Si la propiedad es un método accessor con un getter, y si se especifica el argumento opcional receiver, entonces la función getter se llama como un método de receiver en lugar de como un método de o. Llamar a esta función es similar a evaluar o[name].

#### *Reflect.getOwnPropertyDescriptor(o, name)*

Esta función devuelve un objeto descriptor de propiedades que describe los atributos de la propiedad llamada nombre del objeto o, o devuelve undefined si no existe tal propiedad. Esta función es casi idéntica a `Object.getOwnPropertyDescriptor()`, salvo que la versión de la función de la API de Reflect requiere que el primer argumento sea un objeto y lanza un `TypeError` si no lo es.

#### *Reflect.getPrototypeOf(o)*

Esta función devuelve el prototipo del objeto o o null si el objeto no tiene prototipo. Lanza un `TypeError` si o es un valor primitivo en lugar de un objeto. Esta función es casi idéntica a `Object.getPrototypeOf()` excepto que `Object.getPrototypeOf()` sólo lanza un `TypeError` para argumentos nulos e indefinidos y coacciona otros valores primitivos a sus objetos envolventes.

#### *Reflect.has(o, nombre)*

Esta función devuelve true si el objeto o tiene una propiedad con el nombre especificado (que debe ser una cadena o un símbolo). Llamar a esta función es similar a evaluar el nombre en o.

#### *Reflect.isExtensible(o)*

Esta función devuelve true si el objeto o es extensible ([§14.2](#)) y false si no lo es. Lanza un `TypeError` si o no es un objeto. `Object.isExtensible()` es similar pero simplemente devuelve false cuando se le pasa un argumento que no es un objeto.

#### *Reflect.ownKeys(o)*

Esta función devuelve un array con los nombres de las propiedades del objeto o o lanza un `TypeError` si o no es un objeto. Los nombres de la matriz devuelta serán cadenas y/o símbolos. Llamar a esta función es similar a llamar a `Object.getOwnPropertyNames()` y

`Object.getOwnPropertySymbols()` y combinar sus resultados.

#### *Reflect.preventExtensions(*o*)*

Esta función establece el atributo *extensible* ([§14.2](#)) del objeto *o* a `false` y devuelve `true` para indicar el éxito. Lanza un `TypeError` si *o* no es un objeto.

`Object.preventExtensions()` tiene el mismo efecto pero devuelve `o` en lugar de `true` y no lanza `TypeError` para argumentos que no sean objetos.

#### *Reflect.set(*o*, *nombre*, *valor*, *receptor*)*

Esta función establece la propiedad con el nombre especificado del objeto *o* al valor especificado. Devuelve `true` en caso de éxito y `false` en caso de fallo (que puede ocurrir si la propiedad es de sólo lectura). Lanza `TypeError` si *o* no es un objeto. Si la propiedad especificada es una propiedad accesoria con una función *setter*, y si se pasa el argumento opcional *receiver*, entonces el *setter* será invocado como un método de *receiver* en lugar de ser invocado como un método de *o*. Llamar a esta función es normalmente lo mismo que evaluar *o[name] = value*.

#### *Reflect.setPrototypeOf(*o*, *p*)*

Esta función establece el prototipo del objeto *o* en *p*, devolviendo `true` en caso de éxito y `false` en caso de fallo (lo que puede ocurrir si *o* no es extensible o si la operación causaría una cadena circular de prototipos). Lanza un `TypeError` si *o* no es un objeto o si *p* no es ni un objeto ni `null`. `Object.setPrototypeOf()` es similar, pero devuelve *o* si tiene éxito y lanza un `TypeError` si falla. Recuerde que llamar a cualquiera de estas funciones probablemente hará que su código sea más lento al interrumpir las optimizaciones del intérprete de JavaScript.

## 14.7 Objetos Proxy

La clase `Proxy`, disponible en ES6 y posteriores, es la característica de metaprogramación más potente de JavaScript. Nos permite escribir código que altera el comportamiento fundamental de los objetos de JavaScript. La API de `Reflect` descrita en §14.6 es un conjunto de funciones que nos da acceso directo a un conjunto de operaciones fundamentales sobre objetos de JavaScript. Lo que hace la clase `Proxy` es permitirnos una forma de implementar esas operaciones fundamentales nosotros mismos y crear objetos que se comporten de formas que no son posibles para los objetos ordinarios.

Cuando creamos un objeto `Proxy`, especificamos otros dos objetos, el objeto destino y el objeto manejador:

```
let proxy = new Proxy(target, handlers);
```

El objeto `Proxy` resultante no tiene estado ni comportamiento propio. Cada vez que se realiza una operación en él (leer una propiedad, escribir una propiedad, definir una nueva propiedad, buscar el prototipo, invocarlo como una función), envía esas operaciones al objeto manejador o al objeto destino.

Las operaciones soportadas por los objetos `Proxy` son las mismas que las definidas por la API de `Reflect`. Supongamos que `p` es un objeto `Proxy` y que

`write` delete `p.x`. La función `Reflect.deleteProperty()` tiene el mismo comportamiento que el operador `delete`. Y cuando se utiliza el operador `delete` para eliminar una propiedad de un objeto `Proxy`, busca un método `deleteProperty()` en el objeto `handlers`. Si tal

método existe, lo invoca. Y si no existe tal método, entonces el objeto Proxy realiza el borrado de la propiedad en el objeto destino.

Los proxies funcionan así para todas las operaciones fundamentales: si existe un método apropiado en el objeto manejador, se invoca ese método para realizar la operación. (Los nombres y firmas de los métodos son los mismos que los de las funciones Reflect que se tratan en §14.6.) Y si ese método no existe en el objeto manejador, entonces el Proxy realiza la operación fundamental en el objeto destino. Esto significa que un Proxy puede obtener su comportamiento del objeto destino o del objeto handlers. Si el objeto handlers está vacío, entonces el proxy es esencialmente una envoltura transparente alrededor del objeto objetivo:

```
let t = { x: 1, y: 2 }; let p = new  
Proxy(t, {});  
p.x // => 1 borrar p.y // => true: borrar la propiedad y del proxy  
t.y // => indefinido: esto lo elimina también en el destino  
p.z = 3; // Definir una nueva propiedad en el proxy  
t.z // => 3: define la propiedad en el objetivo
```

Este tipo de proxy envolvente transparente es esencialmente equivalente al objeto de destino subyacente, lo que significa que no hay realmente una razón para utilizarlo en lugar del objeto envuelto. Las envolturas transparentes pueden ser útiles, sin embargo, cuando se crean como "proxies revocables". En lugar de crear un Proxy con el constructor Proxy(), puede utilizar la función de fábrica Proxy.revocable(). Esta función devuelve un objeto que incluye un objeto Proxy y también una función revoke(). Una vez que se llama a la función revoke(), el proxy deja de funcionar inmediatamente:

```
function accessTheDatabase() { /* implementación omitida */ return 42; } let {proxy,  
revoke} = Proxy.revocable(accessTheDatabase, {});
```

```
proxy() // => 42: El proxy da acceso a la función objetivo subyacente revoke(); // Pero ese  
acceso se puede desactivar cuando queramos  
proxy(); // !TypeError: ya no podemos llamar a esta función
```

Tenga en cuenta que además de demostrar los proxies revocables, el código anterior también demuestra que los proxies pueden trabajar con funciones de destino, así como con objetos de destino. Pero el punto principal aquí es que los proxies revocables son un bloque de construcción para un tipo de aislamiento de código, y usted podría utilizarlos cuando se trata de bibliotecas de terceros no confiables, por ejemplo. Si tienes que pasar una función a una biblioteca que no controlas, puedes pasar un proxy revocable en su lugar y luego revocar el proxy cuando hayas terminado con la biblioteca. Esto evita que la biblioteca mantenga una referencia a tu función y la llame en momentos inesperados. Este tipo de programación defensiva no es típica en los programas de JavaScript, pero la clase Proxy al menos lo hace posible.

Si pasamos un objeto handlers no vacío al constructor de Proxy(), entonces ya no estamos definiendo un objeto envolvente transparente y en su lugar estamos implementando un comportamiento personalizado para nuestro proxy. Con el conjunto adecuado de manejadores, el objeto de destino subyacente se vuelve esencialmente irrelevante.

En el siguiente código, por ejemplo, es cómo podríamos implementar un objeto que parezca tener un número infinito de propiedades de sólo lectura, donde el valor de cada propiedad es el mismo que el nombre de la propiedad:

```
// Usamos un Proxy para crear un objeto que parece tener cada
// posible propiedad, con el valor de cada propiedad igual a su nombre let identity =
new Proxy({}, { // Cada propiedad tiene su propio nombre como valor get(o, nombre,
objetivo) { return nombre; }, // Cada nombre de propiedad está definido has(o,
nombre) { return true; },
// Hay demasiadas propiedades para enumerar, así que lanzamos
ownKeys(o) { throw new RangeError("Número infinito de propiedades"); },
// Todas las propiedades existen y no son escribibles, configurables o enumerables.
getOwnPropertyDescriptor(o, name) { devuelve {
valor: name, enumerable: false, escribible: false,
configurable: false };
}, // Todas las propiedades son de sólo lectura por lo que no se pueden
establecer set(o, name, value, target) { return false; },
// Todas las propiedades no son configurables, por lo que no pueden ser eliminadas
deleteProperty(o, name) { return false; },
// Todas las propiedades existen y no son configurables por lo que no podemos
definir más defineProperty(o, name, desc) { return false; },
```

```

// En efecto, esto significa que el objeto no es extensible isExtensible(o)
{ return false; },
// Todas las propiedades ya están definidas en este objeto, por lo que no podría //
heredar nada aunque tuviera un objeto prototipo.
getPrototypeOf(o) { return null; },
// El objeto no es extensible, por lo que no podemos cambiar el prototipo
setPrototypeOf(o, proto) { return false; }, });
}

identity.x //=> "x" identity.toString //=> "toString" identity[0] //=> "0" identity.x = 1;
// La configuración de las propiedades no tiene efecto identity.x //=> "x" delete
identity.x //=> false: tampoco se pueden borrar las propiedades
identidad.x //=> "x"
Object.keys(identity); // !RangeError: can't list all the keys for(let p of identity) ; //
```

*!RangeError*

Los objetos proxy pueden derivar su comportamiento del objeto destino y del objeto manejador, y los ejemplos que hemos visto hasta ahora han utilizado un objeto u otro. Pero normalmente es más útil definir proxies que utilicen ambos objetos.

El siguiente código, por ejemplo, utiliza Proxy para crear una envoltura de sólo lectura para un objeto de destino. Cuando el código intenta leer valores del objeto, esas lecturas son reenviadas al objeto destino normalmente. Pero si algún código intenta modificar el objeto o sus propiedades, los métodos del objeto manejador lanzan un TypeError. Un proxy como este puede ser útil para escribir pruebas: suponga que ha escrito una función que toma un argumento del objeto y quiere asegurarse de que su función no hace ningún intento de modificar el argumento de entrada. Si su

prueba pasa en un objeto envolvente de sólo lectura, entonces cualquier escritura lanzará excepciones que harán que la prueba falle:

```
function readOnlyProxy(o) { function readonly() { throw new TypeError(" Readonly"); }
  return new Proxy(o, { set: readonly, defineProperty: readonly, deleteProperty:
    readonly, setPrototypeOf: readonly, }); }

let o = { x: 1, y: 2 }; // Objeto normal escribible let p = readOnlyProxy(o); //
Versión de sólo lectura
p.x // => 1: la lectura de las propiedades funciona
p.x = 2; // !TypeError: no se pueden cambiar las propiedades delete p.y; //
!TypeError: no se pueden borrar las propiedades
p.z = 3; // !TypeError: no se pueden añadir propiedades
p.__proto__ = {};// !TypeError: no se puede cambiar el prototipo
```

Otra técnica a la hora de escribir proxies es definir métodos manejadores que intercepten operaciones en un objeto pero que sigan delegando las operaciones en el objeto de destino. Las funciones de la API de Reflect ([§14.6](#)) tienen exactamente las mismas firmas que los métodos manejadores, por lo que facilitan ese tipo de delegación.

Aquí, por ejemplo, hay un proxy que delega todas las operaciones en el objeto de destino, pero utiliza métodos manejadores para registrar las operaciones:

```
/*
```

```

*      Devuelve un objeto Proxy que envuelve a o, delegando todas las operaciones a
*      ese objeto después de registrar cada operación. objname es una cadena que
*      aparecerá en los mensajes de registro para identificar el objeto. Ifo tiene su
*      propio
*      cuyos valores son objetos o funciones, entonces si se consulta
*      el valor de esas propiedades, obtendrá un loggingProxy de vuelta, por lo que
*      El comportamiento de registro de este apoderado es "contagioso".
/*/function loggingProxy(o, objname) {
// Definir los manejadores para nuestro objeto Proxy de registro.
// Cada manejador registra un mensaje y luego delega en el objeto de destino.
const handlers = {
  // Este manejador es un caso especial porque para las propiedades propias
  // cuyo valor es un objeto o función, devuelve un proxy en lugar de // devolver el
  valor en sí. get(target, property, receiver) { // Registra la operación get console.
log(`Handler get(${objname},${property}.toString())`);

  // Utilice la API de Reflect para obtener el valor de la propiedad let value = Reflect.
get(target, property, receiver);

  // Si la propiedad es una propiedad propia del objetivo y // el valor es un objeto o
  función entonces devuelve un Proxy para ella.
  if (Reflect.ownKeys(target).includes(property) && (typeof value === "object" || 
typeof value === "function")) { return loggingProxy(value, `${objname}. ${propiedad}.
  toString()}); }
}

// En caso contrario, devuelve el valor sin modificar.
valor de retorno;

```

```

    },

    // Los tres métodos siguientes no tienen nada de especial:      // registran la
    operación y delegan en el objeto de destino.
    // Son un caso especial simplemente para que podamos evitar el registro del
    objeto // receptor que puede causar una recursión infinita.
    set(target, prop, value, receiver) { console.log(`Handler set(${objname},${prop.
    toString()},${value})`); return Reflect.set(target, prop, value, receiver); }, apply(target,
    receiver, args) { console.log(`Handler ${objname}(${args})`); return Reflect.
    apply(target, receiver, args); }, construct(target, args, receiver) { console.log(`Handler
    ${objname}(${args})`); return Reflect.construct(target, args, receiver); }

};

// Podemos generar automáticamente el resto de los manejadores.  //
iMetaprogramación FTW! Reflect.ownKeys(Reflect).forEach(handlerName => { if (!
(handlerName in handlers)) { handlers[handlerName] = function(target, ... args)
{ // Registrar la operación en la consola. log(`Handler ${handlerName}
(${objname},${args})`); // Delegar la operación
    return Reflect[handlerName](target, ... args); };
}

});

// Devuelve un proxy para el objeto usando estos manejadores de registro
return new Proxy(o, handlers);

```

La función `loggingProxy()` definida anteriormente crea proxies que registran todas las formas de uso. Si estás tratando de entender cómo una función no documentada utiliza los objetos que le pasas, usar un proxy de registro puede ayudar.

Considere los siguientes ejemplos, que dan lugar a algunas ideas genuinas sobre la iteración de matrices:

```
// Definir un array de datos y un objeto con una propiedad de función let data = [10,20]; let methods = { square: x => x*x };

// Crear proxies de registro para el array y el objeto let proxyData = loggingProxy(data, "data"); let proxyMethods = loggingProxy(methods, "methods");

// Supongamos que queremos entender cómo funciona el método Array.map() data.map(methods.square) // => [100, 400]

// Primero, probemos con una matriz de Proxy de registro proxyData.map(methods.square) // => [100, 400]
// Produce esta salida:
// Manejador get(data,map)
// Manejador get(datos,longitud)
// Manejador get(datos,constructor)
// Handler has(data,0)
// Manejador get(data,0)
// Handler has(data,1)
// Manejador get(data,1)

// Ahora vamos a probar con un objeto proxy methods data.map(proxyMethods.square) // => [100, 400] // Salida del log:
```

```
// Manejador get(métodos,cuadrado)
// Métodos del manipulador.cuadrado(10,0,10,20)
// Métodos del manipulador.cuadrado(20,1,10,20)

// Por último, usemos un proxy de registro para conocer el protocolo de iteración for(let x of proxyData) console.log("Datum", x);
// Salida del registro:
// Manejador get(data,Symbol(Symbol.iterator))
// Manejador get(datos,longitud)
```

```
// Manejador get(data,0)
// Dato 10
// Manejador get(datos,longitud)
// Manejador get(data,1)
// Dato 20
// Manejador get(datos,longitud)
```

Del primer trozo de salida de registro, aprendemos que el método `Array.map()` comprueba explícitamente la existencia de cada elemento del array (provocando la invocación del manejador `has()`) antes de leer realmente el valor del elemento (que activa el manejador `get()`). Esto es presumiblemente para poder distinguir los elementos del array que no existen de los elementos que existen pero no están definidos.

El segundo trozo de la salida de registro podría recordarnos que la función que pasamos a `Array.map()` se invoca con tres argumentos: el valor del elemento, el índice del elemento y el propio array. (Hay un problema en nuestra salida de registro: el método `Array.toString()` no incluye corchetes en su salida, y los mensajes de registro serían más claros si se incluyeran en la lista de argumentos (10,0, [10,20]).)

El tercer trozo de salida del registro nos muestra que el bucle `for/of` funciona buscando un método con nombre simbólico `[Symbol.iterator]`. También demuestra que la implementación de la clase `Array` de este método iterador tiene cuidado de comprobar la longitud del array en cada iteración y no asume que la longitud del array permanece constante durante la iteración.

### 14.7.1 Invariantes de Proxy

La función `readOnlyProxy()` definida anteriormente crea objetos `Proxy` que están efectivamente congelados: cualquier intento de alterar un valor de propiedad o un atributo de propiedad o de añadir o eliminar propiedades lanzará una excepción. Pero mientras el objeto de destino no esté congelado, encontraremos que si podemos consultar el proxy con `Reflect.isExtensible()` y `Reflect.getOwnPropertyDescriptor()`, y nos dirá que deberíamos poder establecer, añadir y eliminar propiedades. Así que `readOnlyProxy()` crea objetos en un estado inconsistente. Podríamos arreglar esto añadiendo manejadores `isExtensible()` y `getOwnPropertyDescriptor()`, o podemos vivir con este tipo de inconsistencia menor.

Sin embargo, la API del manejador `Proxy` nos permite definir objetos con inconsistencias mayores, y en este caso, la propia clase `Proxy` nos impedirá crear objetos `Proxy` que sean inconsistentes de mala manera. Al principio de esta sección, describimos a los proxies como objetos sin comportamiento propio porque simplemente reenvían todas las operaciones al objeto manejador y al objeto destino. Pero esto no es del todo cierto: después de reenviar una operación, la clase `Proxy` realiza algunas comprobaciones de sanidad en el resultado para asegurar que no se violan invariantes importantes de JavaScript. Si detecta una violación, el proxy lanzará un `TypeError` en lugar de permitir que la operación continúe.

Por ejemplo, si se crea un proxy para un objeto no extensible, el proxy lanzará un `TypeError` si el manejador `isExtensible()` devuelve alguna vez `true`:

```
let target = Object.preventExtensions({});  
let proxy = new Proxy(target, { isExtensible() { return true; }});
```

```
Reflect.isExtensible(proxy); // !TypeError: violación de invariante
```

En relación con esto, los objetos proxy para objetivos no extensibles no pueden tener un manejador `getPrototypeOf()` que devuelva otra cosa que no sea el objeto prototipo real del objetivo. Además, si el objeto de destino tiene propiedades no escribibles y no configurables, la clase `Proxy` lanzará un `TypeError` si el manejador `get()` devuelve cualquier cosa que no sea el valor real:

```
let target = Object.freeze({x: 1}); let proxy = new Proxy(target, { get() { return 99; } }); proxy.x; // !TypeError: el valor devuelto por get() no coincide con el target
```

El proxy impone una serie de invariantes adicionales, casi todas ellas relacionadas con los objetos de destino no extensibles y las propiedades no configurables del objeto de destino.

## 14.8 Resumen

En este capítulo has aprendido:

- Los objetos de JavaScript tienen un atributo *extensible* y las propiedades de los objetos tienen atributos *escribibles*, *enumerables* y *configurables*, así como un valor y un atributo getter y/o setter. Puedes utilizar estos atributos para "bloquear" tus objetos de varias maneras, incluyendo la creación de objetos "sellados" y "congelados".
- JavaScript define funciones que le permiten recorrer la cadena de prototipos de un objeto e incluso cambiar el prototipo de un objeto (aunque hacer esto puede hacer que su código sea más lento).
- Las propiedades del objeto `Symbol` tienen valores que son "símbolos conocidos", que puede utilizar como nombres de propiedades o métodos para los objetos y clases que defina. Esto le

---

permite controlar la forma en que su objeto interactúa con las características del lenguaje JavaScript y con la biblioteca central. Por ejemplo, los símbolos conocidos le permiten hacer que sus clases sean iterables y controlar la cadena que se muestra cuando se pasa una instancia a `Object.prototype.toString()`.

Antes de ES6, este tipo de personalización sólo estaba disponible para las clases nativas que se incorporaban a una implementación.

- Los literales de plantilla etiquetados son una sintaxis de invocación de funciones, y definir una nueva función de etiqueta es como añadir una nueva sintaxis literal al lenguaje. Definir una función de etiqueta que analice su argumento de cadena de plantilla permite incrustar DSL en el código JavaScript. Las funciones de etiqueta también proporcionan acceso a una forma cruda, sin esconder, de literales de cadena donde las barras invertidas no tienen un significado especial.
- La clase `Proxy` y la API `Reflect` relacionada permiten un control de bajo nivel sobre los comportamientos fundamentales de los objetos de JavaScript. Los objetos proxy pueden utilizarse como envolturas opcionalmente revocables para mejorar la encapsulación del código, y también pueden utilizarse para implementar comportamientos de objetos no estándar (como algunas de las API de casos especiales definidas por los primeros navegadores web).

Un error en el motor V8 de JavaScript hace que este código no funcione correctamente en Node



# Capítulo 15. JavaScript en los navegadores web

---

El lenguaje JavaScript se creó en 1994 con el propósito expreso de permitir un comportamiento dinámico en los documentos mostrados por los navegadores web. El lenguaje ha evolucionado considerablemente desde entonces, y al mismo tiempo, el alcance y las capacidades de la plataforma web han crecido de forma explosiva. Hoy en día, los programadores de JavaScript pueden pensar en la web como una plataforma completa para el desarrollo de aplicaciones. Los navegadores web están especializados en la visualización de texto e imágenes formateadas, pero, al igual que los sistemas operativos nativos, los navegadores también ofrecen otros servicios, como gráficos, vídeo, audio, redes, almacenamiento e hilos. JavaScript es el lenguaje que permite a las aplicaciones web utilizar los servicios proporcionados por la plataforma web, y este capítulo demuestra cómo se pueden utilizar los más importantes de estos servicios.

El capítulo comienza con el modelo de programación de la plataforma web, explicando cómo se incrustan los scripts dentro de las páginas HTML ([§15.1](#)) y cómo el código JavaScript se activa de forma asíncrona mediante eventos ([§15.2](#)). Las secciones que siguen a este material introductorio documentan las principales API de JavaScript que permiten a las aplicaciones web:

- Controlar el contenido del documento ([§15.3](#)) y el estilo
- ([§15.4](#))

Determinar la posición en pantalla de los elementos del documento ([§15.5](#))

- Crear componentes de interfaz de usuario reutilizables ([§15.6](#))
- Dibujar gráficos ([§15.7](#) y [§15.8](#))
- Reproducir y generar sonidos ([§15.9](#))
- Gestionar la navegación y el historial del navegador ([§15.10](#))
- Intercambio de datos a través de la red ([§15.11](#))
- Almacenar los datos en el ordenador del usuario [§15.12](#))
- Realizar cálculos concurrentes con hilos ([§15.13](#))

#### JAVASCRIPT DEL LADO DEL

E este libro, y en la web, verás el término "JavaScript del lado del cliente". El término es simplemente un sinónimo de JavaScript escrito para ejecutarse en un navegador web, y se contrapone al código "del lado del servidor", que se ejecuta en los servidores

Los dos "lados" se refieren a los dos extremos de la conexión de red que separan el servidor web y el desarrollo de software para la web suele requerir que el código se escriba en ambos "lados". El lado del cliente y el lado del servidor también suelen llamarse "frontend" y "backend".

Las ediciones anteriores de este libro intentaban cubrir de forma exhaustiva todas las APIs de JavaScript definidas por los navegadores web, y como resultado, este libro era demasiado largo hace una década. El número y la complejidad de las API de la web han seguido creciendo, y ya no creo que tenga sentido intentar cubrirlas todas en un solo libro. A partir de la séptima edición, mi objetivo es cubrir definitivamente el lenguaje JavaScript y proporcionar una introducción en profundidad al uso del lenguaje con Node y con los navegadores web. Este capítulo no puede cubrir todas las APIs web, pero presenta las más importantes con suficiente detalle como para

que puedas empezar a utilizarlas de inmediato. Y, habiendo aprendido sobre las APIs principales cubiertas aquí, deberías ser capaz de recoger nuevas APIs (como las resumidas en §15.15) cuando y si las necesitas.

## LEGACY APIs

En los 25 años transcurridos desde el lanzamiento de JavaScript, los proveedores de navegadores han ido añadiendo funciones y API para que los programadores las utilicen. Muchas de esas APIs son ahora obsoletas. Entre ellas se encuentran:

- APIs propietarias que nunca fueron estandarizadas y/o nunca fueron implementadas por otros proveedores de navegadores. El Internet Explorer de Microsoft definió muchas de estas APIs. Algunas (como la propiedad innerHTML) demostraron ser útiles y finalmente fueron estandarizadas. Otras (como el método attachEvent()) han estado obsoletas durante años.
- APIs ineficientes (como el método document.write()) que tienen un impacto tan severo en el rendimiento que su uso ya no se considera aceptable.
- APIs obsoletas que hace tiempo han sido sustituidas por nuevas APIs para conseguir lo mismo. Un ejemplo es document.bgColor, que se definió para permitir a JavaScript establecer el color de fondo de un documento. Con la llegada de CSS, document.bgColor se convirtió en un pionero caso especial sin ningún propósito real.
- APIs mal diseñadas que han sido sustituidas por otras mejores. En los primeros días de la web, los comités de estándares definieron la API clave del Modelo de Objetos de Documentos de forma agnóstica, de modo que la misma API pudiera utilizarse en programas Java para trabajar con documentos XML y en programas JavaScript para trabajar con documentos HTML. El resultado fue una API que no se adaptaba bien al lenguaje JavaScript y que tenía características que no interesaban especialmente a los programadores web. Se tardó décadas en recuperarse de esos primeros errores de diseño, pero los navegadores web actuales soportan un Modelo de Objetos de Documento muy mejorado.

Es posible que los proveedores de navegadores necesiten dar soporte a estas APIs heredadas en un futuro previsible para garantizar la compatibilidad con versiones anteriores, pero ya no es necesario que este libro las documente o que tú aprendas sobre ellas. La plataforma web ha madurado y se ha estabilizado, y si eres un desarrollador web experimentado que recuerda la cuarta o quinta edición de este libro, puede que tengas tantos conocimientos obsoletos que olvidar como material nuevo que aprender.

Node tiene una única implementación y una única fuente autorizada de documentación. Las APIs de la web, por el contrario, se definen por consenso entre los principales proveedores de navegadores web, y la documentación autorizada adopta la forma de una especificación destinada a los programadores de C++ que implementan la API, no a los programadores de JavaScript que la utilizarán. Afortunadamente, el [proyecto "MDN web docs" de Mozilla](#) es una fuente fiable y completa<sup>1</sup> de documentación sobre la API web.

## 15.1 Fundamentos de la programación web

Esta sección explica cómo se estructuran los programas JavaScript para la web, cómo se cargan en un navegador web, cómo obtienen la entrada, cómo producen la salida y cómo se ejecutan de forma asíncrona respondiendo a eventos.

### 15.1.1 JavaScript en las etiquetas HTML <script>

Los navegadores web muestran documentos HTML. Si quieres que un navegador web ejecute código JavaScript, debes incluir (o referenciar) ese código desde un documento HTML, y esto es lo que hace la etiqueta HTML <script>.

El código JavaScript puede aparecer en línea dentro de un archivo HTML entre las etiquetas <script> y </script>. Aquí, por ejemplo, hay un archivo HTML que incluye una etiqueta script con código JavaScript que actualiza dinámicamente un elemento del documento para que se comporte como un reloj digital:

```
<!DOCTYPE html> <!-- Este es un archivo HTML5 -->
<html> <!-- El elemento raíz -->
<head> <!-- El título, los scripts y los estilos pueden ir aquí -->
<título> Reloj digital</título>
<style> /* Una hoja de estilos CSS para el reloj */ #clock { /* Los estilos se aplican al
elemento con id="clock" */ font: bold 24px sans-serif; /* Utiliza una fuente grande en
negrita */ background: #ddf; /* sobre un fondo gris azulado claro. */ padding: 15px;
/* Rodéalo con algo de espacio */
```

```
border: solid black 2px; /* y un borde negro sólido */ border-radius: 10px; /* con
esquinas redondeadas. */
} </style>
</head> <body> <!-- El cuerpo contiene el contenido del documento. -->
```

```
<h1>Reloj digital</h1> <p>Muestra un título. -->
<span id="clock"></span> <!-- Insertaremos la hora en este elemento. --> <script> //
Definir una función para mostrar la hora actual function displayTime() {
    let clock = document.querySelector("#clock"); // Obtener el elemento con id="clock"
    let now = new Date(); // Obtener la hora actual
    clock.textContent = now.toLocaleTimeString(); // Mostrar la hora en el reloj }
displayTime() // Mostrar la hora de inmediato
setInterval(displayTime, 1000); // Y luego actualizarlo cada segundo. </script>
</body>
</html>
```

Aunque el código JavaScript puede incrustarse directamente dentro de una etiqueta `<script>`, es más común utilizar el atributo `src` de la etiqueta `<script>` para especificar la URL (una URL absoluta o una URL relativa a la URL del archivo HTML que se está mostrando) de un archivo que contiene código JavaScript. Si sacamos el código JavaScript de este archivo HTML y lo almacenamos en su propio archivo `scripts/digital_clock.js`, entonces la etiqueta `<script>` podría hacer referencia a ese archivo de código de la siguiente manera:

```
<script src="scripts/digital_clock.js"></script>
```

Un archivo de JavaScript contiene puro JavaScript, sin etiquetas `<script>` ni ningún otro HTML. Por convención, los archivos de código JavaScript tienen nombres que terminan en `.js`.

Una etiqueta `<script>` con el atributo `src` se comporta exactamente como si el contenido del archivo JavaScript especificado apareciera directamente entre las etiquetas `<script>` y `</script>`. Tenga en cuenta que la etiqueta `</script>` de cierre es necesaria en los documentos HTML incluso cuando se especifica el atributo `src`: HTML no admite la etiqueta `<script/>`.

El uso del atributo `src` tiene varias ventajas:

- Simplifica sus archivos HTML al permitirle eliminar grandes bloques de código JavaScript de ellos, es decir, ayuda a mantener separados el contenido y el comportamiento.
- Cuando varias páginas web comparten el mismo código JavaScript, el uso del atributo src permite mantener una sola copia de ese código, en lugar de tener que editar cada archivo HTML cuando el código cambia.
- Si un archivo de código JavaScript es compartido por más de una página, sólo tiene que ser descargado una vez, por la primera página que lo utiliza; las páginas posteriores pueden recuperarlo de la caché del navegador.
- Como el atributo src toma como valor una URL arbitraria, un programa JavaScript o una página web de un servidor web puede emplear código exportado por otros servidores web. Gran parte de la publicidad en Internet se basa en este hecho.

## MÓDULOS

§10.3 documenta los módulos de JavaScript y cubre sus directivas de importación y exportación. Si ha escrito su programa de JavaScript utilizando módulos (y no ha utilizado una herramienta de agrupación de código para combinar todos sus módulos en un único archivo de JavaScript no modular), entonces debe cargar el módulo de nivel superior de su programa con una etiqueta `<script>` que tenga un atributo `type="module"`. Si hace esto, se cargará el módulo que especifique, y se cargarán todos los módulos que importe, y (recursivamente) todos los módulos que importen. Para más detalles, véase §10.3.5.

## ESPECIFICAR EL TIPO DE GUIÓN

En los primeros días de la web, se pensaba que los navegadores podrían implementar algún día lenguajes distintos de JavaScript, y

los programadores añadían atributos como `language="javascript"` y `type="application/javascript"` a sus etiquetas `<script>`.

Esto es completamente innecesario. JavaScript es el lenguaje por defecto (y único) de la web. El atributo `language` está obsoleto, y sólo hay dos razones para utilizar un atributo `type` en una etiqueta `<script>`:

Para • especificar que el script es un módulo

Para • incrustar datos en una página web sin mostrarlos (véase [§15.3.4](#))

## CUANDO SE EJECUTAN LOS SCRIPTS: ASYNC Y DIFERIDO

Cuando se incorporó JavaScript a los navegadores web, no existía una API para recorrer y manipular la estructura y el contenido de un documento ya renderizado. La única forma en que el código JavaScript podía afectar al contenido de un documento era generando ese contenido sobre la marcha mientras el documento estaba en proceso de carga. Para ello, se utilizaba el método `document.write()` para injectar texto HTML en el documento en la ubicación del script.

El uso de `document.write()` ya no se considera un buen estilo, pero el hecho de que sea posible significa que cuando el analizador HTML se encuentra con un elemento `<script>`, debe, por defecto, ejecutar el script sólo para asegurarse de que no sale ningún HTML antes de que pueda reanudar el análisis y la representación del documento. Esto puede ralentizar drásticamente el análisis sintáctico y la representación de la página web.

Afortunadamente, este modo de ejecución de scripts *sincrónico* o de *bloqueo* por defecto no es la única opción. La etiqueta `<script>`

puede tener atributos de aplazamiento y asíncrono, que hacen que los scripts se ejecuten de forma diferente. Se trata de atributos booleanos: no tienen un valor; sólo tienen que estar presentes en la etiqueta <script>. Ten en cuenta que estos atributos sólo tienen sentido cuando se utilizan junto con el atributo src:

```
<script deferido src="deferred.js"></script>
<script async src="async.js"></script>
```

Ambos atributos, defer y async, son formas de decirle al navegador que el script vinculado no utiliza document.write() para generar la salida HTML, y que el navegador, por lo tanto, puede continuar analizando y renderizando el documento mientras se descarga el script. El atributo defer hace que el navegador aplace la ejecución del script hasta que el documento haya sido completamente cargado y analizado y esté listo para ser manipulado. El atributo async hace que el navegador ejecute el script tan pronto como sea posible, pero no bloquea el análisis del documento mientras se descarga el script. Si una etiqueta <script> tiene ambos atributos, el atributo async tiene prioridad.

Tenga en cuenta que los scripts diferidos se ejecutan en el orden en que aparecen en el documento. Los scripts asíncronos se ejecutan a medida que se cargan, lo que significa que pueden ejecutarse fuera de orden.

Los scripts con el atributo type="module" se ejecutan, por defecto, después de que el documento se haya cargado, como si tuvieran el atributo defer. Puedes anular este valor por defecto con el atributo async, que hará que el código se ejecute tan pronto como el módulo y todas sus dependencias se hayan cargado.

Una alternativa simple a los atributos `async` y `defer` -especialmente para el código que se incluye directamente en el HTML- es simplemente poner sus scripts al final del archivo HTML. De esta manera, el script puede ejecutarse sabiendo que el contenido del documento anterior ha sido analizado y está listo para ser manipulado.

## CARGA DE SCRIPTS BAJO DEMANDA

A veces, puede tener código JavaScript que no se utiliza cuando un documento se carga por primera vez y que sólo se necesita si el usuario realiza alguna acción como pulsar un botón o abrir un menú. Si está desarrollando su código utilizando módulos, puede cargar un módulo bajo demanda con `import()`, como se describe en [§10.3.6](#).

Si no está utilizando módulos, puede cargar un archivo de JavaScript a petición simplemente añadiendo una etiqueta `<script>` a su documento cuando quiera que se cargue el script:

```
// Cargar y ejecutar de forma asíncrona un script desde un
// URL
// Devuelve una Promise que resuelve cuando el script se ha cargado.
function importScript(url) { return new Promise((resolve, reject)
  => {
    let s = document.createElement("script"); // Crear un elemento <script>.
    s.onload = () => { resolve(); }; // Resuelve la promesa cuando se carga
    s.onerror = (e) => { reject(e); }; // Rechazar en caso de fallo
    s.src = url; // Establecer la URL del script
    document.head.append(s); // Añadir <script> al documento
  });
}
```

Esta función importScript() utiliza las APIs del DOM ([§15.3](#)) para crear una nueva etiqueta <script> y añadirla al documento <head>. Y utiliza manejadores de eventos ([§15.2](#)) para determinar cuándo el script se ha cargado con éxito o cuándo ha fallado la carga.

### 15.1.2 El modelo de objetos del documento

Uno de los objetos más importantes en la programación JavaScript del lado del cliente es el objeto Document, que representa el documento HTML que se muestra en una ventana o pestaña del navegador. La API para trabajar con documentos HTML se conoce como el Modelo de Objetos de Documento, o DOM, y se cubre en detalle en [§15.3](#). Pero el DOM es tan importante para la programación JavaScript del lado del cliente que merece ser introducido aquí.

Los documentos HTML contienen elementos HTML anidados unos dentro de otros, formando un árbol. Considere el siguiente documento HTML simple:

```
<html>
<cabeza>
  <título> Documento de muestra</título>
</cabeza>
<cuerpo>
  <h1> Un documento HTML</h1> <p>Este es un documento <i>simple</i>.
</cuerpo>
</html>
```

La etiqueta <html> de nivel superior contiene las etiquetas <head> y <body>. La etiqueta

La etiqueta <head> contiene una etiqueta <title>. Y la etiqueta <body> contiene las etiquetas <h1> y <p>. Las etiquetas <title> y

`<h1>` contienen cadenas de texto, y la etiqueta `<p>` contiene dos cadenas de texto con una etiqueta `<i>` entre ellas.

La API DOM refleja la estructura de árbol de un documento HTML. Para cada etiqueta HTML en el documento, hay un objeto Elemento JavaScript correspondiente, y para cada tramo de texto en el documento, hay un objeto Texto correspondiente. Las clases Element y Text, así como la propia clase Document, son todas subclases de la clase Node, más general, y los objetos Node están organizados en una estructura de árbol que JavaScript puede consultar y recorrer utilizando la API DOM. La representación DOM de este documento es el árbol que se muestra en la [Figura 15-1](#).

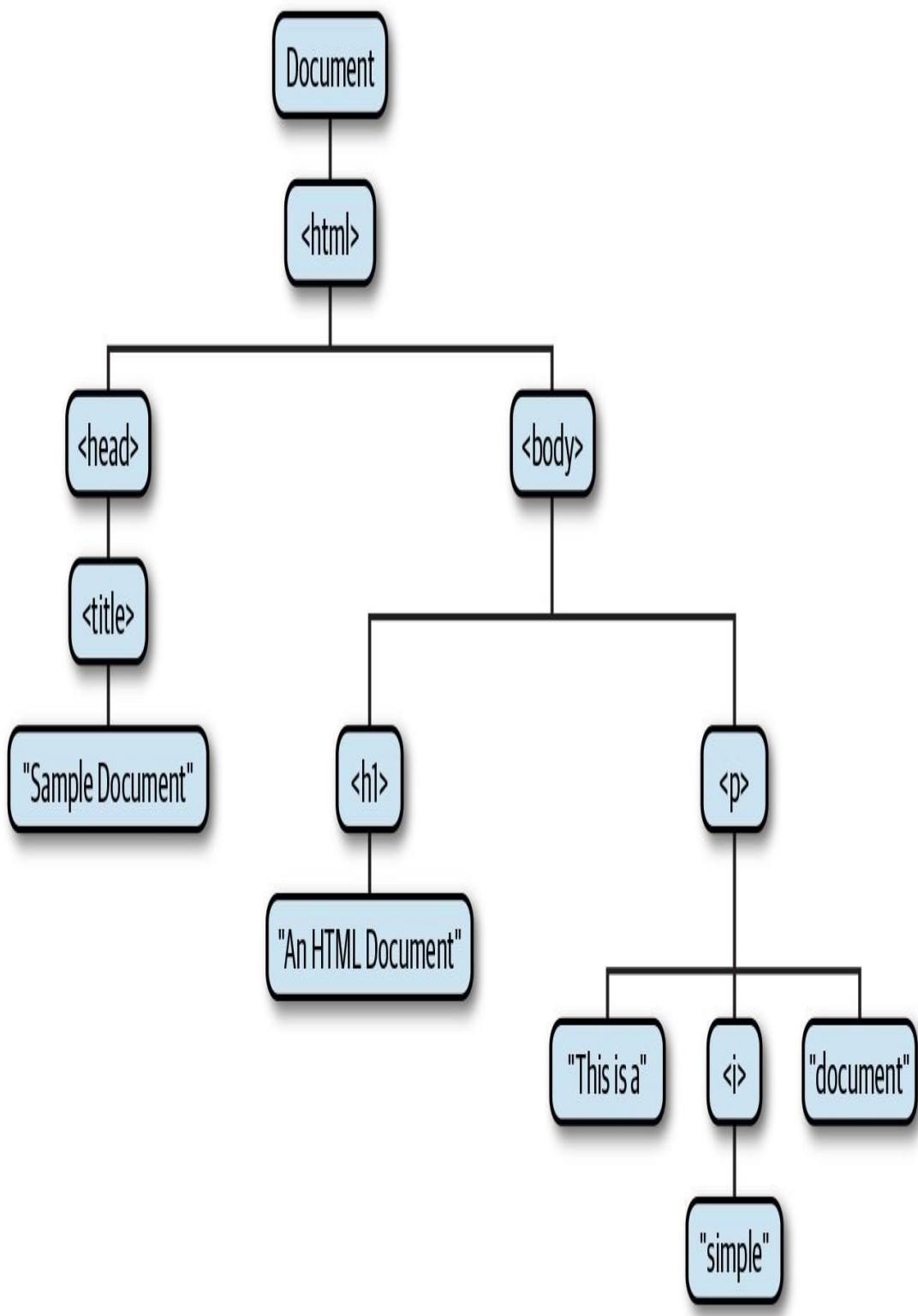


Figura 15-1. La representación en forma de árbol de un documento HTML Si aún no está familiarizado con las estructuras de árbol en la programación informática, es útil saber que toman prestada

la terminología de los árboles genealógicos. El nodo que está directamente encima de un nodo es el *padre* de ese nodo. Los nodos que están un nivel por debajo de otro nodo son los *hijos* de ese nodo. Los nodos del mismo nivel y con el mismo padre son *hermanos*. El conjunto de nodos situados un número cualquiera de niveles por debajo de otro nodo son los *descendientes de ese nodo*. Y el padre, el abuelo y todos los demás nodos por encima de un nodo son los *antepasados de ese nodo*.

La API del DOM incluye métodos para crear nuevos nodos de Elemento y Texto, y para insertarlos en el documento como hijos de otros objetos Elemento. También hay métodos para mover elementos dentro del documento y para eliminarlos por completo. Mientras que una aplicación del lado del servidor puede producir una salida de texto plano escribiendo cadenas con `console.log()`, una aplicación JavaScript del lado del cliente puede producir una salida HTML formateada construyendo o manipulando el documento del árbol de documentos utilizando la API DOM.

Hay una clase de JavaScript que corresponde a cada tipo de etiqueta HTML, y cada aparición de la etiqueta en un documento está representada por una instancia de la clase. La etiqueta `<body>`, por ejemplo, está representada por una instancia de `HTMLBodyElement`, y una etiqueta `<table>` está representada por una instancia de `HTMLTableElement`. Los objetos de elementos de JavaScript tienen propiedades que corresponden a los atributos HTML de las etiquetas.

Por ejemplo, las instancias de `HTMLImageElement`, que representan las etiquetas `<img>`, tienen una propiedad `src` que se corresponde con el atributo `src` de la etiqueta. El valor inicial de la propiedad `src` es el valor del atributo que aparece en la etiqueta HTML, y el establecimiento de esta propiedad con JavaScript cambia el valor del atributo HTML (y hace que el navegador cargue y muestre una nueva imagen). La mayoría de las clases de elementos de JavaScript sólo reflejan los atributos de una etiqueta HTML, pero algunas definen métodos adicionales. Las clases `HTMLAudioElement` y `HTMLVideoElement`, por ejemplo, definen métodos como `play()` y `pause()` para controlar la reproducción de archivos de audio y vídeo.

### 15.1.3 El objeto global en los navegadores web

Hay un objeto global por cada ventana o pestaña del navegador ([§3.7](#)). Todo el código JavaScript (excepto el código que se ejecuta en hilos de trabajo; véase [§15.13](#)) que se ejecuta en esa ventana comparte este único objeto global. Esto es cierto independientemente de cuántos scripts o módulos haya en el documento: todos los scripts y módulos de un documento comparten un único objeto global; si un script define una propiedad en ese objeto, esa propiedad es visible también para todos los demás scripts.

El objeto global es donde se define la biblioteca estándar de JavaScript: la función `parseInt()`, el objeto `Math`, la clase `Set`, etc. En los navegadores web, el objeto global también contiene los principales puntos de entrada de varias APIs web. Por ejemplo, la propiedad `document` representa el documento que se muestra actualmente, el método `fetch()` realiza peticiones de red HTTP y el

constructor Audio() permite a los programas de JavaScript reproducir sonidos.

En los navegadores web, el objeto global tiene una doble función: además de definir los tipos y funciones incorporados, también representa la ventana actual del navegador web y define propiedades como history ([§15.10.2](#)), que representa el historial de navegación de la ventana, e innerWidth, que contiene el ancho de la ventana en píxeles. Una de las propiedades de este objeto global se llama window, y su valor es el propio objeto global. Esto significa que puedes simplemente escribir window para referirte al objeto global en tu código del lado del cliente. Cuando se utilizan características específicas de la ventana, a menudo es una buena idea incluir un prefijo window.innerWidth es más claro que innerWidth, por ejemplo.

#### **15.1.4 Los scripts comparten un espacio de nombres**

Con los módulos, las constantes, variables, funciones y clases definidas en el nivel superior (es decir, fuera de cualquier función o definición de clase) del módulo son privadas para el módulo a menos que se exporten explícitamente, en cuyo caso, pueden ser importadas selectivamente por otros módulos. (Tenga en cuenta que esta propiedad de los módulos también es respetada por las herramientas de agrupación de código).

Sin embargo, con los scripts que no son módulos, la situación es completamente diferente. Si el código de nivel superior de un script define una constante, variable, función o clase, esa declaración será visible para todos los demás scripts del mismo documento. Si un script define una función f() y otro script define una clase c, entonces

un tercer script puede invocar la función e instanciar la clase sin tener que realizar ninguna acción para importarlos. Por lo tanto, si no está utilizando módulos, los scripts independientes en su documento comparten un único espacio de nombres y se comportan como si fueran todos parte de un único script más grande. Esto puede ser conveniente para programas pequeños, pero la necesidad de evitar conflictos de nombres puede ser problemática para programas más grandes, especialmente cuando algunos de los scripts son bibliotecas de terceros.

Hay algunas peculiaridades históricas en el funcionamiento de este espacio de nombres compartido. Las declaraciones de var y de función en el nivel superior crean propiedades en el objeto global compartido. Si un script define una función de nivel superior f(), entonces otro script en el mismo documento puede invocar esa función como f() o como window.f(). Por otro lado, las declaraciones ES6 const, let y class, cuando se usan en el nivel superior, no crean propiedades en el objeto global. Sin embargo, siguen estando definidas en un espacio de nombres compartido: si un script define una clase C, otros scripts podrán crear instancias de esa clase con new C(), pero no con new window.C().

Resumiendo: en los módulos, las declaraciones de nivel superior se circunscriben al módulo y pueden exportarse explícitamente. En los scripts que no son módulos, sin embargo, las declaraciones de nivel superior se refieren al documento que los contiene, y las declaraciones son compartidas por todos los scripts del documento. Las declaraciones var y function más antiguas se comparten a través de las propiedades del objeto global. Las declaraciones const, let y class más recientes también se comparten y tienen el mismo ámbito

del documento, pero no existen como propiedades de ningún objeto al que el código JavaScript tenga acceso.

### 15.1.5 Ejecución de programas JavaScript

No existe una definición formal de un *programa* en el lado del cliente de JavaScript, pero podemos decir que un programa de JavaScript consiste en todo el código de JavaScript en, o referenciado desde, un documento. Estos trozos de código separados comparten un único objeto Window global, que les da acceso al mismo objeto Document subyacente que representa el documento HTML. Los scripts que no son módulos comparten además un espacio de nombres de nivel superior.

Si una página web incluye un marco incrustado (utilizando el elemento <iframe>), el código JavaScript en el documento incrustado tiene un objeto global y un objeto Documento diferentes a los del código en el documento incrustado, y puede considerarse un programa JavaScript independiente. Recuerde, sin embargo, que no existe una definición formal de cuáles son los límites de un programa JavaScript. Si el documento contenedor y el documento contenido se cargan desde el mismo servidor, el código de un documento puede interactuar con el código del otro, y puede tratarlos como dos partes interactivas de un único programa, si lo desea. En §15.13.6 se explica cómo un programa JavaScript puede enviar y recibir mensajes hacia y desde el código JavaScript que se ejecuta en un <iframe>.

Se puede pensar que la ejecución de un programa JavaScript se produce en dos fases. En la primera fase, se carga el contenido del documento y se ejecuta el código de los elementos <script> (tanto los

scripts en línea como los externos). Los scripts generalmente se ejecutan en el orden en que aparecen en el documento, aunque este orden por defecto puede ser modificado por los atributos `async` y `defer` que hemos descrito. El código JavaScript dentro de cualquier script se ejecuta de arriba a abajo, sujeto, por supuesto, a los condicionales, bucles y otras declaraciones de control de JavaScript. Algunos scripts no *hacen* realmente nada durante esta primera fase y se limitan a definir funciones y clases para su uso en la segunda fase. Otros scripts pueden hacer un trabajo significativo durante la primera fase y luego no hacer nada en la segunda. Imagina un script al final de un documento que encuentra todas las etiquetas `<h1>` y `<h2>` en el documento y modifica el documento generando e insertando una tabla de contenidos al principio del documento. Esto podría hacerse completamente en la primera fase. (Véase en §15.3.6 un ejemplo que hace exactamente esto).

Una vez cargado el documento y ejecutados todos los scripts, la ejecución de JavaScript entra en su segunda fase. Esta fase es asíncrona y dirigida por eventos. Si un script va a participar en esta segunda fase, entonces una de las cosas que debe haber hecho durante la primera fase es registrar al menos un manejador de eventos u otra función de devolución de llamada que será invocada asincrónicamente. Durante esta segunda fase dirigida por eventos, el navegador web invoca funciones manejadoras de eventos y otras llamadas de retorno en respuesta a eventos que ocurren asincrónicamente. Los manejadores de eventos se invocan más comúnmente en respuesta a la entrada del usuario (clics del ratón, pulsaciones de teclas, etc.), pero también pueden ser activados por la actividad de la red, la carga de documentos y recursos, el tiempo transcurrido o los errores en el código.

JavaScript. Los eventos y los manejadores de eventos se describen en detalle en §15.2.

Algunos de los primeros eventos que se producen durante la fase dirigida por eventos son los eventos "DOMContentLoaded" y "load". El evento "DOMContentLoaded" se activa cuando el documento HTML ha sido completamente cargado y analizado. El evento "load" se activa cuando todos los recursos externos del documento -como las imágenes- también se han cargado por completo. Los programas de JavaScript suelen utilizar uno de estos eventos como disparador o señal de inicio. Es común ver programas cuyos scripts definen funciones pero no realizan ninguna acción más que registrar una función manejadora de eventos para ser disparada por el evento "load" al comienzo de la fase de ejecución dirigida por eventos. Es este manejador de eventos "load" el que luego manipula el documento y hace lo que sea que el programa deba hacer. Tenga en cuenta que es común en la programación de JavaScript que una función manejadora de eventos como el manejador de eventos "load" descrito aquí registre otros manejadores de eventos.

La fase de carga de un programa JavaScript es relativamente corta: idealmente menos de un segundo. Una vez cargado el documento, la fase basada en eventos dura tanto tiempo como el documento sea mostrado por el navegador web. Debido a que esta fase es asíncrona y basada en eventos, puede haber largos períodos de inactividad en los que no se ejecuta JavaScript, interrumpidos por ráfagas de actividad desencadenadas por eventos del usuario o de la red.

A continuación trataremos estas dos fases con más detalle.

## MODELO DE HILOS DEL LADO DEL CLIENTE EN JAVASCRIPT

JavaScript es un lenguaje de un solo hilo, y la ejecución de un solo hilo hace que la programación sea mucho más sencilla: puedes escribir código con la seguridad de que dos manejadores de eventos nunca se ejecutarán al mismo tiempo. Puedes manipular el contenido de un documento sabiendo que ningún otro hilo está intentando modificarlo al mismo tiempo, y nunca tendrás que preocuparte por bloqueos, puntos muertos o condiciones de carrera al escribir código JavaScript.

La ejecución de un solo hilo significa que los navegadores web dejan de responder a la entrada del usuario mientras se ejecutan los scripts y los manejadores de eventos. Esto supone una carga para los programadores de JavaScript: significa que los scripts de JavaScript y los manejadores de eventos no deben ejecutarse durante demasiado tiempo. Si un script realiza una tarea intensiva de cálculo, introducirá un retraso en la carga del documento, y el usuario no verá el contenido del documento hasta que el script finalice. Si un manejador de eventos realiza una tarea computacionalmente intensiva, el navegador puede dejar de responder, posiblemente haciendo que el usuario piense que se ha colapsado.

La plataforma web define una forma controlada de concurrencia llamada "web worker". Un web worker es un hilo en segundo plano para realizar tareas computacionalmente intensivas sin congelar la interfaz de usuario. El código que se ejecuta en un hilo web worker no tiene acceso a

contenido del documento, no comparte ningún estado con el hilo principal o con otros workers, y sólo puede comunicarse con el hilo

principal y otros workers a través de eventos de mensajes asíncronos, por lo que la concurrencia no es detectable para el hilo principal, y los web workers no alteran el modelo básico de ejecución de un solo hilo de los programas JavaScript. Véase [§15.13](#) para conocer todos los detalles sobre el mecanismo de roscado seguro de la web.

## LÍNEA DE TIEMPO JAVASCRIPT DEL LADO DEL CLIENTE

Ya hemos visto que los programas de JavaScript comienzan en una fase de ejecución de scripts y luego pasan a una fase de manejo de eventos. Estas dos fases pueden dividirse en los siguientes pasos:

1. El navegador web crea un objeto Document y comienza a analizar la página web, añadiendo objetos Element y nodos Text al documento a medida que analiza los elementos HTML y su contenido textual. La propiedad document.readyState tiene el valor "loading" en esta etapa.
2. Cuando el analizador HTML encuentra una etiqueta <script> que no tiene ninguno de los atributos async, defer, o type="module", añade esa etiqueta script al documento y luego ejecuta el script. El script se ejecuta de forma sincrónica, y el analizador HTML hace una pausa mientras el script se descarga (si es necesario) y se ejecuta. Un script como este puede usar document.write() para insertar texto en el flujo de entrada, y ese texto se convertirá en parte del documento cuando el analizador se reanude. Un script como este a menudo simplemente define funciones y registra manejadores de eventos para su uso posterior, pero puede atravesar y manipular el árbol del documento tal y como existe en ese momento. Es decir, los scripts no modulares que no tienen un atributo async o defer pueden ver su propia etiqueta <script> y el contenido del documento que le precede.

3. Cuando el analizador encuentra un elemento <script> que tiene el atributo `async` establecido, comienza a descargar el texto del script (y si el script es un módulo, también descarga recursivamente todas las dependencias del script) y continúa analizando el documento. El script se ejecutará tan pronto como sea posible después de que se haya descargado, pero el analizador no se detiene y espera a que se descargue. Los scripts asíncronos no deben utilizar el método `document.write()`. Pueden ver su propia etiqueta <script> y todo el contenido del documento que le precede, y pueden o no tener acceso al contenido adicional del documento.
4. Cuando el documento está completamente analizado, la propiedad `document.readyState` cambia a "interactive".
5. Todos los scripts que tenían el atributo de aplazamiento establecido (junto con cualquier script de módulo que no tenga un atributo `async`) se ejecutan en el orden en que aparecieron en el documento. Los scripts asíncronos también pueden ejecutarse en este momento. Los scripts diferidos tienen acceso al documento completo y no deben utilizar el método `document.write()`.
6. El navegador lanza un evento "DOMContentLoaded" en el objeto Document. Esto marca la transición de la fase sincrónica de ejecución de scripts a la fase asíncrona, basada en eventos, de la ejecución del programa. Tenga en cuenta, sin embargo, que todavía puede haber scripts asíncronos que aún no se han ejecutado en este punto.
7. El documento está completamente analizado en este punto, pero el navegador puede seguir esperando a que se carguen contenidos adicionales, como las imágenes. Cuando todo este contenido termina de cargarse, y cuando todos los scripts asíncronos se han cargado y ejecutado, la propiedad

`document.readyState` cambia a "complete" y el navegador web lanza un evento "load" en la ventana objeto.

8. A partir de este punto, los manejadores de eventos se invocan de forma asíncrona en respuesta a eventos de entrada del usuario, eventos de red, vencimiento de temporizadores, etc.

### 15.1.6 Entrada y salida del programa

Como cualquier programa, los programas JavaScript del lado del cliente procesan datos de entrada para producir datos de salida.

Hay una variedad de entradas disponibles:

- El contenido del propio documento, al que el código JavaScript puede acceder con la API DOM ([§15.3](#)).
- La entrada del usuario, en forma de eventos, como los clics del ratón (o los toques de la pantalla táctil) en elementos HTML `<button>`, o el texto introducido en elementos HTML `<textarea>`, por ejemplo. En [§15.2](#) se muestra cómo los programas de JavaScript pueden responder a este tipo de eventos del usuario.
- La URL del documento que se muestra está disponible para el JavaScript del lado del cliente como `document.URL`. Si pasas esta cadena al constructor de `URL()` ([§11.9](#)), puedes acceder fácilmente a las secciones de ruta, consulta y fragmento de la URL.
- El contenido de la cabecera de petición HTTP "Cookie" está disponible para el código del lado del cliente como `document.cookie`. Las cookies suelen ser utilizadas por el código del lado del servidor para mantener las sesiones del usuario, pero el código del lado del cliente también puede leerlas (y escribirlas) si es necesario. Véase [§15.12.2](#) para más detalles.

- La propiedad global `navigator` proporciona acceso a información sobre el navegador web, el sistema operativo sobre el que se ejecuta y las capacidades de cada uno. Por ejemplo, `navigator.userAgent` es una cadena que identifica el navegador web, `navigator.language` es el idioma preferido del usuario, y `navigator.hardwareConcurrency` devuelve el número de CPUs lógicas disponibles para el navegador web. Del mismo modo, la propiedad global `screen` proporciona acceso al tamaño de la pantalla del usuario a través de las propiedades `screen.width` y `screen.height`. En cierto sentido, estos objetos navegador y pantalla son para los navegadores web lo que las variables de entorno son para los programas Node.

El JavaScript del lado del cliente suele producir salida, cuando lo necesita, manipulando el documento HTML con la API DOM ([§15.3](#)) o utilizando un marco de trabajo de nivel superior como React o Angular para manipular el documento. El código del lado del cliente también puede usar `console.log()` y métodos relacionados ([§11.8](#)) para producir una salida. Pero esta salida sólo es visible en la consola del desarrollador web, por lo que es útil para la depuración, pero no para la salida visible para el usuario.

### 15.1.7 Errores del programa

A diferencia de las aplicaciones (como las aplicaciones Node) que se ejecutan directamente sobre el sistema operativo, los programas JavaScript en un navegador web no pueden realmente "colapsar". Si se produce una excepción mientras su programa JavaScript se está ejecutando, y si no tiene una sentencia `catch` para manejarla, se mostrará un mensaje de error en la consola del desarrollador, pero cualquier controlador de eventos que se haya registrado sigue ejecutándose y respondiendo a los eventos.

Si desea definir un manejador de errores de último recurso para ser invocado cuando se produzca este tipo de excepción no capturada, establezca la propiedad `onerror` del objeto `Window` a una función manejadora de errores. Cuando una excepción no capturada se propaga por toda la pila de llamadas y un mensaje de error está a punto de ser mostrado en la consola del desarrollador, la función `window.onerror` será invocada con tres argumentos de cadena. El primer argumento de `window.onerror` es un mensaje que describe el error. El segundo argumento es una cadena que contiene la URL del código JavaScript que ha causado el error. El tercer argumento es el número de línea dentro del documento donde se produjo el error. Si el manejador `onerror` devuelve `true`, le dice al navegador que el manejador ha manejado el error y que no es necesaria ninguna otra acción—en otras palabras, el navegador no debe mostrar su propio mensaje de error.

Cuando una `Promise` es rechazada y no hay una función `.catch()` para manejarla, es una situación muy parecida a una excepción no manejada: un error no anticipado o un error lógico en tu programa. Puedes detectar esto definiendo una función `window.onunhandledrejection` o utilizando `window.addEventListener()` para registrar un manejador de eventos "unhandledrejection". El objeto de evento pasado a este manejador tendrá una propiedad `promise` cuyo valor es el objeto `Promise` que rechazó y una propiedad `reason` cuyo valor es lo que se habría pasado a una función `.catch()`. Al igual que con los manejadores de error descritos anteriormente, si se llama a `preventDefault()` en el objeto de evento de rechazo no manejado, se considerará manejado y no causará un mensaje de error en la consola del desarrollador.

No suele ser necesario definir onerror o onunhandledrejection, pero puede ser bastante útil como mecanismo de telemetría si quieras informar de los errores del lado del cliente al servidor (usando la función fetch() para hacer una petición HTTP POST, por ejemplo) para que puedas obtener información sobre los errores inesperados que ocurren en los navegadores de tus usuarios.

### **15.1.8 El modelo de seguridad web**

El hecho de que las páginas web puedan ejecutar código JavaScript arbitrario en su dispositivo personal tiene claras implicaciones de seguridad, y los proveedores de navegadores han trabajado duro para equilibrar dos objetivos que compiten entre sí:

- Definición de potentes APIs del lado del cliente para permitir aplicaciones web útiles
- Evitar que un código malicioso lea o altere sus datos, comprometa su privacidad, le estafe o le haga perder el tiempo

Las subsecciones que siguen dan una rápida visión de las restricciones y problemas de seguridad que usted, como programador de JavaScript, debe conocer.

#### **LO QUE NO PUEDE HACER JAVASCRIPT**

La primera línea de defensa de los navegadores web contra el código malicioso es que simplemente no soportan ciertas capacidades. Por ejemplo, el JavaScript del lado del cliente no proporciona ninguna forma de escribir o eliminar archivos arbitrarios o listar directorios arbitrarios en el ordenador del cliente.

Esto significa que un programa JavaScript no puede borrar datos ni plantar virus.

Del mismo modo, el JavaScript del lado del cliente no tiene capacidades de red de propósito general. Un programa JavaScript del lado del cliente puede hacer peticiones HTTP ([§15.11.1](#)). Y otro estándar, conocido como WebSockets ([§15.11.3](#)), define una API tipo socket para comunicarse con servidores especializados. Pero ninguna de estas APIs permite el acceso sin intermediarios a la red más amplia. Los clientes y servidores de Internet de uso general no pueden escribirse en JavaScript del lado del cliente.

## LA POLÍTICA DEL MISMO ORIGEN

La *política del mismo origen* es una restricción de seguridad muy amplia sobre el contenido web con el que puede interactuar el código JavaScript. Suele entrar en juego cuando una página web incluye elementos <iframe>. En este caso, la política del mismo origen rige las interacciones del código JavaScript en un marco con el contenido de otros marcos. En concreto, un script sólo puede leer las propiedades de las ventanas y documentos que tienen el mismo origen que el documento que contiene el script.

El origen de un documento se define como el protocolo, el host y el puerto de la URL desde la que se cargó el documento. Los documentos cargados desde diferentes servidores web tienen diferentes orígenes. Los documentos cargados a través de diferentes puertos del mismo host tienen diferentes orígenes. Y un documento cargado con el protocolo http: tiene un origen diferente al de uno cargado con el protocolo https:, aunque provengan del mismo servidor web. Los navegadores suelen tratar

cada archivo: URL como un origen separado, lo que significa que si está trabajando en un programa que muestra más de un documento del mismo servidor, es posible que no pueda probarlo localmente usando file: URL y tendrá que ejecutar un servidor web estático durante el desarrollo.

Es importante entender que el origen del script en sí mismo no es relevante para la política del mismo origen: lo que importa es el origen del documento en el que el script está incrustado.

Supongamos, por ejemplo, que un script alojado en el host A se incluye (utilizando la propiedad src de un elemento <script>) en una página web servida por el host B. El origen de ese script es el host B, y el script tiene pleno acceso al contenido del documento que lo contiene. Si el documento contiene un <iframe> que contiene un segundo documento del host B, entonces el script también tiene acceso completo al contenido de ese segundo documento. Pero si el documento de nivel superior contiene otro <iframe> que muestra un documento del host C (o incluso uno del host A), entonces la política del mismo origen entra en vigor e impide que el script acceda a este documento anidado.

La política del mismo origen también se aplica a las peticiones HTTP con scripts (véase §15.11.1). El código JavaScript puede hacer peticiones HTTP arbitrarias al servidor web desde el que se cargó el documento que lo contiene, pero no permite que los scripts se comuniquen con otros servidores web (a menos que esos servidores web opten por CORS, como describimos a continuación).

La política del mismo origen plantea problemas para los sitios web grandes que utilizan varios subdominios. Por ejemplo, los scripts con origen *órdenes.ejemplo.com* pueden necesitar leer propiedades de documentos en *ejemplo.com*. Para soportar sitios web multidominio de este tipo, los scripts pueden alterar su origen estableciendo `document.domain` a un sufijo de dominio. Así, un script con origen <https://orders.example.com> puede cambiar su origen a <https://example.com> estableciendo `document.domain` como "ejemplo.com". Pero ese script no puede establecer `document.domain` a "pedidos.ejemplo", "amplio.com" o "com".

La segunda técnica para flexibilizar la política del mismo origen es CrossOrigin Resource Sharing, o CORS, que permite a los servidores decidir qué orígenes están dispuestos a servir. CORS amplía HTTP con una nueva cabecera de solicitud `Origin`: y una nueva cabecera de respuesta `Access-Control-Allow-Origin`. Permite a los servidores utilizar una cabecera para enumerar explícitamente los orígenes que pueden solicitar un archivo o utilizar un comodín y permitir que un archivo sea solicitado por cualquier sitio. Los navegadores respetan estas cabeceras CORS y no relajan las restricciones del mismo origen a menos que estén presentes.

## SECUENCIAS DE COMANDOS EN EL SITIO WEB

*Cross-site scripting*, o XSS, es un término para una categoría de problemas de seguridad en la que un atacante inyecta etiquetas HTML o scripts en un sitio web de destino. Los programadores de JavaScript del lado del cliente deben conocer y defenderse del cross-site scripting.

Una página web es vulnerable al cross-site scripting si genera dinámicamente el contenido del documento y basa ese contenido en datos enviados por el usuario sin "sanear" primero esos datos eliminando cualquier etiqueta HTML incrustada en ellos. Como ejemplo trivial, considere la siguiente página web que utiliza JavaScript para saludar al usuario por su nombre:

```
<script>
let name = new URL(document. URL). searchParams. get("name"); document.
querySelector('h1'). innerHTML = "Hello " + name; </script>
```

Esta secuencia de comandos de dos líneas extrae la información del parámetro de consulta "nombre" de la URL del documento. A continuación, utiliza la API del DOM para injectar una cadena HTML en la primera etiqueta `<h1>` del documento. Esta página está pensada para ser invocada con una URL como esta:

<http://www.example.com/greet.html?name=David>

Cuando se utiliza así, muestra el texto "Hola David". Pero considere lo que sucede cuando se invoca con este parámetro de consulta:

`name=%3Cimg%20src=%22x.png%22%20onload=%22alert(%27hacked%27) %22/%3E`

Cuando se decodifican los parámetros de la URL, esta URL hace que se inyecte el siguiente HTML en el documento:

Hola 

Después de cargar la imagen, se ejecuta la cadena de JavaScript en el atributo onload. La función global alert() muestra un cuadro de diálogo modal. Un solo cuadro de diálogo es relativamente benigno,

pero demuestra que la ejecución de código arbitrario es posible en este sitio porque muestra HTML sin desinfectar.

Los ataques de scripting entre sitios se llaman así porque hay más de un sitio implicado. El sitio B incluye un enlace especialmente diseñado (como el del ejemplo anterior) hacia el sitio A. Si el sitio B puede convencer a los usuarios de que hagan clic en el enlace, serán llevados al sitio A, pero ese sitio estará ahora ejecutando código del sitio B. Ese código podría desfigurar la página o hacer que funcione mal. Y lo que es más peligroso, el código malicioso podría leer las cookies almacenadas por el sitio A (quizás números de cuenta u otra información de identificación personal) y enviar esos datos de vuelta al sitio B. El código injectado podría incluso rastrear las pulsaciones del usuario y enviar esos datos de vuelta al sitio B.

En general, la forma de prevenir los ataques XSS es eliminar las etiquetas HTML de cualquier dato que no sea de confianza antes de utilizarlo para crear el contenido del documento dinámico. Puede arreglar el archivo *greet.html* mostrado anteriormente sustituyendo los caracteres HTML especiales de la cadena de entrada no fiable por sus entidades HTML equivalentes:

```
nombre = nombre . replace(/&/g,  
"&")  
. replace(/</g, "<")  
. replace(/>/g, ">")  
. replace(/\"/g, """)  
. replace(/\'/g, "'")  
. replace(/\//g, "&#x2F;")
```

Otro enfoque del problema del XSS es estructurar sus aplicaciones web de manera que el contenido no fiable se muestre siempre en

un <iframe> con el atributo sandbox configurado para deshabilitar el scripting y otras capacidades.

El cross-site scripting es una vulnerabilidad perniciosa cuyas raíces se hunden en la arquitectura de la web. Merece la pena conocer esta vulnerabilidad en profundidad, pero su análisis va más allá del alcance de este libro. Hay muchos recursos en línea que le ayudarán a defenderse contra el cross-site scripting.

## 15.2 Eventos

Los programas JavaScript del lado del cliente utilizan un modelo de programación asíncrono basado en eventos. En este estilo de programación, el navegador web genera un *evento* cada vez que ocurre algo interesante en el documento o en el navegador o en algún elemento u objeto asociado a él. Por ejemplo, el navegador web genera un evento cuando termina de cargar un documento, cuando el usuario mueve el ratón sobre un hipervínculo, o cuando el usuario pulsa una tecla en el teclado. Si una aplicación JavaScript se preocupa por un tipo de evento en particular, puede registrar una o más funciones para que sean invocadas cuando ocurran eventos de ese tipo. Tenga en cuenta que esto no es exclusivo de la programación web: todas las aplicaciones con interfaces gráficas de usuario están diseñadas de esta manera: se sientan a esperar que se interactúe con ellas (es decir, esperan a que se produzcan eventos) y luego responden.

En JavaScript del lado del cliente, los eventos pueden ocurrir en cualquier elemento dentro de un documento HTML, y este hecho hace que el modelo de eventos de los navegadores web sea

significativamente más complejo que el modelo de eventos de Node. Comenzamos esta sección con algunas definiciones importantes que ayudan a explicar ese modelo de eventos:

#### *tipo de evento*

Esta cadena especifica el tipo de evento ocurrido. El tipo "mousemove", por ejemplo, significa que el usuario movió el ratón. El tipo "keydown" significa que el usuario pulsó una tecla del teclado hacia abajo. Y el tipo "load" significa que un documento (o algún otro recurso) ha terminado de cargarse desde la red. Como el tipo de un evento es sólo una cadena, a veces se le llama *nombre del evento*, y de hecho, usamos este nombre para identificar el tipo de evento del que estamos hablando.

#### *objetivo del evento*

Es el objeto en el que se ha producido el evento o con el que se asocia el evento. Cuando hablamos de un evento, debemos especificar tanto el tipo como el objetivo. Un evento de carga en una Ventana, por ejemplo, o un evento de clic en un Elemento <button>. Los objetos Window, Document y Element son los objetivos de eventos más comunes en las aplicaciones JavaScript del lado del cliente, pero algunos eventos se activan en otros tipos de objetos. Por ejemplo, un objeto Worker (un tipo de hilo, cubierto en [§15.13](#)) es un objetivo para los eventos "mensaje" que ocurren cuando el hilo worker envía un mensaje al hilo principal.

#### *manejador de eventos, o escuchador de eventos*

Esta función maneja o responde a un evento.<sup>2</sup> Las aplicaciones registran sus funciones manejadoras de eventos con el navegador web, especificando un tipo de evento y un objetivo de evento. Cuando un evento del tipo especificado ocurre en el objetivo especificado, el navegador invoca la función manejadora. Cuando los manejadores de eventos son invocados para un objeto, decimos que el navegador ha "disparado" o "despachado" el evento. Hay varias maneras de registrar los manejadores de eventos, y los detalles del

registro e invocación de los manejadores se explican en §15.2.2 y §15.2.3.

### *objeto del evento*

Este objeto está asociado a un evento concreto y contiene detalles sobre el mismo. Los objetos de evento se pasan como argumento a la función del manejador de eventos. Todos los objetos de evento tienen una propiedad type que especifica el tipo de evento y una propiedad target que especifica el objetivo del evento. Cada tipo de evento define un conjunto de propiedades para su objeto de evento asociado. El objeto asociado a un evento de ratón incluye las coordenadas del puntero del ratón, por ejemplo, y el objeto asociado a un evento de teclado contiene detalles sobre la tecla que se pulsó y las teclas modificadoras que se mantuvieron pulsadas. Muchos tipos de eventos definen sólo unas pocas propiedades estándar -como el tipo y el objetivo- y no llevan mucha otra información útil. Para esos eventos, lo que importa es la simple ocurrencia del evento, no los detalles del mismo.

### *propagación de eventos*

Este es el proceso por el cual el navegador decide en qué objetos se activan los manejadores de eventos. Para los eventos que son específicos de un solo objeto -como el evento "load" en el objeto Window o un evento "message" en un objeto Worker- no se requiere propagación.

Sin embargo, cuando ciertos tipos de eventos ocurren en elementos dentro del documento HTML, se propagan o "burbujean" hacia arriba en el árbol del documento. Si el usuario mueve el ratón sobre un hipervínculo, el evento mousemove se dispara primero en el elemento <a> que define ese enlace. Luego se dispara en los elementos que lo contienen: quizás un elemento <p>, un elemento <section>, y el propio objeto Document. A veces es más conveniente registrar un único manejador de eventos en un Documento u otro elemento contenedor que registrar manejadores en cada elemento individual que te interesa. Un manejador de eventos puede detener la propagación de un evento para que no continúe burbujeando y no

dispare manejadores en los elementos que lo contienen. Los manejadores hacen esto invocando un método del objeto de evento. En otra forma de propagación de eventos, conocida como *captura de eventos*, los manejadores especialmente registrados en elementos contenedores tienen la oportunidad de interceptar (o "capturar") eventos antes de que sean entregados a su objetivo real. El burbujeo y la captura de eventos se tratan en detalle en [§15.2.4](#).

Algunos eventos tienen *acciones por defecto* asociadas a ellos. Cuando se produce un evento de clic en un hipervínculo, por ejemplo, la acción por defecto es que el navegador siga el enlace y cargue una nueva página. Los manejadores de eventos pueden evitar esta acción por defecto invocando un método del objeto del evento.

Esto se llama a veces "cancelar" el evento y se trata en [el §15.2.5](#).

### 15.2.1 Categorías de eventos

El JavaScript del lado del cliente soporta un número tan grande de tipos de eventos que es imposible que este capítulo los cubra todos. Sin embargo, puede ser útil agrupar los eventos en algunas categorías generales, para ilustrar el alcance y la amplia variedad de eventos soportados:

#### *Eventos de entrada dependientes del dispositivo*

Estos eventos están directamente vinculados a un dispositivo de entrada específico, como el ratón o el teclado. Incluyen tipos de eventos como "mousedown", "mousemove", "mouseup", "touchstart", "touchmove", "touchend", "keydown" y "keyup".

#### *Eventos de entrada independientes del dispositivo*

Estos eventos de entrada no están directamente vinculados a un dispositivo de entrada específico. El evento "clic", por ejemplo, indica que se ha activado un enlace o un botón (u otro elemento del documento). Esto se hace a menudo mediante un clic del ratón, pero también podría hacerse mediante el teclado o (en dispositivos táctiles) con un toque. El evento "input" es una alternativa independiente del dispositivo al evento "keydown" y admite la entrada por teclado, así como alternativas como cortar y pegar y métodos de entrada utilizados para scripts ideográficos. Los eventos "pointerdown", "pointermove" y "pointerup" son alternativas independientes del dispositivo a los eventos de ratón y táctiles. Funcionan para los punteros de tipo ratón, para las pantallas táctiles y también para la entrada de tipo lápiz o stylus.

### *Eventos de la interfaz de usuario*

Los eventos UI son eventos de alto nivel, a menudo en elementos de formularios HTML que definen una interfaz de usuario para una aplicación web. Incluyen el evento "focus" (cuando un campo de entrada de texto obtiene el foco del teclado), el evento "change" (cuando el usuario cambia el valor mostrado por un elemento de formulario) y el evento "submit" (cuando el usuario hace clic en un botón de envío en un formulario).

### *Eventos de cambio de estado*

Algunos eventos no se desencadenan directamente por la actividad del usuario, sino por la actividad de la red o del navegador, e indican algún tipo de cambio relacionado con el ciclo de vida o el estado. Los eventos "load" y "DOMContentLoaded" -disparados en los objetos Window y Document, respectivamente, al final de la carga del documento- son probablemente los más utilizados de estos eventos (ver "[Línea de tiempo JavaScript del lado del cliente](#)"). Los navegadores disparan eventos "online" y "offline" en el objeto Window cuando cambia la conectividad de la red. El mecanismo de gestión del historial del navegador ([§15.10.4](#)) dispara el evento "popstate" en respuesta al botón Atrás del navegador.

### *Eventos específicos de la API*

Una serie de APIs web definidas por HTML y especificaciones relacionadas incluyen sus propios tipos de eventos. Los elementos HTML <video> y <audio> definen una larga lista de tipos de eventos asociados, como "espera", "reproducción", "búsqueda", "cambio de volumen", etc., y se pueden utilizar para personalizar la reproducción de medios. En general, las APIs de plataformas web que son asíncronas y se desarrollaron antes de que se añadieran las promesas a JavaScript están basadas en eventos y definen eventos específicos de la API. La API IndexedDB, por ejemplo ([§15.12.3](#)), dispara eventos de "éxito" y "error" cuando las peticiones a la base de datos tienen éxito o fallan. Y aunque la nueva API `fetch()` ([§15.11.1](#)) para realizar peticiones HTTP está basada en promesas, la API XMLHttpRequest a la que sustituye define una serie de tipos de eventos específicos de la API.

## 15.2.2 Registro de manejadores de eventos

Hay dos formas básicas de registrar los manejadores de eventos. La primera, desde los primeros días de la web, es establecer una propiedad en el objeto o elemento del documento que es el objetivo del evento. La segunda técnica (más reciente y general) es pasar el controlador al método `addEventListener()` del objeto o elemento.

### ESTABLECER LAS PROPIEDADES DE LOS MANEJADORES DE EVENTOS

La forma más sencilla de registrar un manejador de eventos es estableciendo una propiedad del objetivo del evento a la función deseada del manejador de eventos. Por convención, las propiedades de los manejadores de eventos tienen nombres que consisten en la palabra "on" seguida del nombre del evento: `onclick`, `onchange`, `onload`, `onmouseover`, etc. Tenga en cuenta que los nombres de estas propiedades distinguen entre mayúsculas y minúsculas,<sup>3</sup> incluso cuando el tipo de evento (como "mousedown") consta de varias

palabras. El siguiente código incluye dos registros de manejadores de eventos de este tipo:

```
// Establece la propiedad onload del objeto Window en una función. // La función es el  
manejador de eventos: se invoca cuando se carga el documento.  
window.onload = function() { // Buscar un elemento <form> let form =  
document.querySelector("form#shipping");  
    // Registrar una función manejadora de eventos en el formulario que será invocada  
    // antes de que el formulario sea enviado. Asumir que isValid() está definido en  
    // otro lugar. form.onSubmit = function(event) { // Cuando el usuario envía el formulario  
if (!isValid(this)) { // comprueba si las entradas del formulario son válidas event.  
preventDefault(); // y si no, impide el envío del formulario. }  
};  
};
```

El defecto de las propiedades de los manejadores de eventos es que están diseñadas bajo la suposición de que los objetivos de eventos tendrán como máximo un manejador para cada tipo de evento. A menudo es mejor registrar los manejadores de eventos usando addEventListener() porque esa técnica no sobrescribe ningún manejador previamente registrado.

## ESTABLECER LOS ATRIBUTOS DE LOS MANEJADORES DE EVENTOS

Las propiedades de los manejadores de eventos de los elementos del documento también pueden definirse directamente en el archivo HTML como atributos en la etiqueta HTML correspondiente. (Los manejadores que se registrarían en el elemento Window con JavaScript pueden definirse con atributos en la etiqueta <body> en HTML). Esta técnica es generalmente mal vista en el desarrollo web

moderno, pero es posible, y se documenta aquí porque todavía se puede ver en el código existente.

Cuando se define un manejador de eventos como un atributo HTML, el valor del atributo debe ser una cadena de código JavaScript. Ese código debe ser el *cuerpo* de la función del manejador de eventos, no una declaración de función completa. Es decir, el código HTML del controlador de eventos no debe estar rodeado de llaves y precedido por la palabra clave function. Por ejemplo:

```
<button onclick="console.log('Gracias');"> Por favor  
Clic</botón>
```

Si un atributo de controlador de eventos de HTML contiene varias sentencias de JavaScript, debe recordar separar esas sentencias con punto y coma o dividir el valor del atributo en varias líneas.

Cuando se especifica una cadena de código JavaScript como el valor de un atributo manejador de eventos HTML, el navegador convierte su cadena en una función que funciona algo así:

```
function(event) { with(document) { with(this. form || {}) { with(this) { /* su código aquí */  
}}}  
}  
}  
}
```

El argumento event significa que el código de su manejador puede referirse al objeto de evento actual como event. Las sentencias with significan que el código de tu manejador puede referirse a las propiedades del objeto destino, el <form> contenedor (si lo hay) y el objeto Documento contenedor directamente, como si fueran variables en el ámbito. La sentencia with está prohibida en modo

estricto ([§5.6.3](#)), pero el código JavaScript en los atributos HTML nunca es estricto. Los manejadores de eventos definidos de este modo se ejecutan en un entorno en el que se definen variables inesperadas. Esto puede ser una fuente de errores confusos y es una buena razón para evitar escribir manejadores de eventos en HTML.

## ADDEVENTLISTENER()

Cualquier objeto que pueda ser un objetivo de evento-esto incluye los objetos Window y Document y todos los Elementos del documento-define un método llamado addEventListener() que puedes usar para registrar un manejador de eventos para ese objetivo. addEventListener() toma tres argumentos. El primero es el tipo de evento para el que se registra el controlador. El tipo de evento (o nombre) es una cadena que no incluye el prefijo "on" utilizado cuando se establecen las propiedades del manejador de eventos. El segundo argumento de addEventListener() es la función que debe ser invocada cuando ocurre el tipo de evento especificado. El tercer argumento es opcional y se explica a continuación.

El siguiente código registra dos manejadores para el evento "click" en un elemento <button>. Observa las diferencias entre las dos técnicas utilizadas:

```
<button id="mybutton"> Haz clic en mí</button>
<script>
let b = document.querySelector("#mybutton");
b.onclick = function() { console.log("¡Gracias por hacer clic en mí!"); };
b.addEventListener("click", () => { console.log("¡Gracias de nuevo!"); });
</script>
```

Llamar a addEventListener() con "click" como primer argumento no afecta al valor de la propiedad onclick. En este código, un clic de

botón registrará dos mensajes en la consola del desarrollador. Y si llamáramos primero a addEventListener() y luego estableciéramos onclick, seguiríamos registrando dos mensajes, sólo que en el orden inverso. Más importante aún, puedes llamar a addEventListener() varias veces para registrar más de una función manejadora para el mismo tipo de evento en el mismo objeto. Cuando un evento ocurre en un objeto, todos los manejadores registrados para ese tipo de evento son invocados en el orden en que fueron registrados. Invocar addEventListener() más de una vez en el mismo objeto con los mismos argumentos no tiene ningún efecto - la función manejadora permanece registrada sólo una vez, y la invocación repetida no altera el orden en que se invocan los manejadores.

addEventListener() está emparejado con un removeEventListener() que espera los mismos dos argumentos (más un tercero opcional) pero que elimina una función manejadora de eventos de un objeto en lugar de añadirla. A menudo es útil registrar temporalmente un controlador de eventos y eliminarlo poco después. Por ejemplo, cuando recibes un evento "mousedown", puedes registrar temporalmente manejadores de eventos para "mousemove" y "mouseup" para ver si el usuario arrastra el ratón.

A continuación, desegistrarías estos manejadores cuando llegue el evento "mouseup". En esta situación, el código de eliminación de los manejadores de eventos podría ser así:

```
document.removeEventListener("mousemove", handleMouseMove); document.  
removeEventListener("mouseup", handleMouseUp);
```

El tercer argumento opcional de `addEventListener()` es un valor booleano o un objeto. Si pasas `true`, entonces tu función manejadora es registrada como manejadora de eventos de captura y es invocada en una fase diferente del despacho de eventos.

Cubriremos la captura de eventos en §15.2.4. Si pasas un tercer argumento de `true` cuando registras un manejador de eventos, entonces también debes pasar `true` como tercer argumento a `removeEventListener()` si quieras eliminar el manejador.

Registrar un manejador de eventos de captura es sólo una de las tres opciones que admite `addEventListener()`, y en lugar de pasar un único valor booleano, también puede pasar un objeto que especifique explícitamente las opciones que desea:

```
document.addEventListener("click", handleClick, { capture: true, once: true, passive: true });
```

Si el objeto Options tiene una propiedad de captura establecida en `true`, entonces el manejador de eventos será registrado como un manejador de captura. Si esa propiedad es falsa o se omite, entonces el manejador será de no captura.

Si el objeto Options tiene la propiedad `once` establecida en `true`, entonces el manejador de eventos se eliminará automáticamente después de que se dispare una vez. Si esta propiedad es falsa o se omite, entonces el manejador nunca se elimina automáticamente.

Si el objeto Options tiene una propiedad pasiva establecida en `true`, indica que el manejador de eventos nunca llamará a `preventDefault()` para cancelar la acción por defecto (ver §15.2.5). Esto es particularmente importante para los eventos táctiles en

dispositivos móviles-si los manejadores de eventos "touchmove" pueden prevenir la acción de desplazamiento por defecto del navegador, entonces el navegador no puede implementar el desplazamiento suave. Esta propiedad pasiva proporciona una forma de registrar un manejador de eventos potencialmente disruptivo de este tipo, pero permite al navegador web saber que puede comenzar con seguridad su comportamiento por defecto -como el desplazamiento- mientras el manejador de eventos se está ejecutando. El desplazamiento suave es tan importante para una buena experiencia de usuario que Firefox y Chrome hacen que los eventos "touchmove" y "mousewheel" sean pasivos por defecto. Así que si quieras registrar un manejador que llame a preventDefault() para uno de estos eventos, debes establecer explícitamente la propiedad passive a false.

También se puede pasar un objeto Options a removeEventListener(), pero la propiedad capture es la única relevante. No es necesario especificar once o passive cuando se elimina un listener, y estas propiedades se ignoran.

### **15.2.3 Invocación del manejador de eventos**

Una vez que hayas registrado un manejador de eventos, el navegador web lo invocará automáticamente cuando se produzca un evento del tipo especificado en el objeto especificado. Esta sección describe la invocación de manejadores de eventos en detalle, explicando los argumentos de los manejadores de eventos, el contexto de invocación (el valor this), y el significado del valor de retorno de un manejador de eventos.

## ARGUMENTO DEL MANEJADOR DE EVENTOS

Los manejadores de eventos se invocan con un objeto Evento como único argumento. Las propiedades del objeto Evento proporcionan detalles sobre el evento:

### *tipo*

El tipo de evento que se produjo.

### *objetivo*

El objeto en el que se produjo el evento.

### *currentTarget*

Para los eventos que se propagan, esta propiedad es el objeto en el que se registró el manejador de eventos actual.

### *timeStamp*

Una marca de tiempo (en milisegundos) que representa cuándo ocurrió el evento pero que no representa un tiempo absoluto. Puedes determinar el tiempo transcurrido entre dos eventos restando la marca de tiempo del primer evento de la marca de tiempo del segundo.

### *isTrusted*

Esta propiedad será verdadera si el evento fue despachado por el propio navegador web y falsa si el evento fue despachado por código JavaScript.

Algunos tipos de eventos tienen propiedades adicionales. Los eventos de ratón y puntero, por ejemplo, tienen propiedades clientX y clientY que especifican las coordenadas de la ventana en la que se produjo el evento.

## CONTEXTO DEL MANEJADOR DE EVENTOS

Cuando se registra un manejador de eventos estableciendo una propiedad, parece que se está definiendo un nuevo método en el objeto de destino:

```
target.onclick = function() { /* handler code */};
```

No es sorprendente, por tanto, que los manejadores de eventos sean invocados como métodos del objeto sobre el que se definen. Es decir, dentro del cuerpo de un manejador de eventos, la palabra clave `this` se refiere al objeto sobre el que se registró el manejador de eventos.

Los manejadores son invocados con el objetivo como su valor `this`, incluso cuando son registrados usando `addEventListener()`. Sin embargo, esto no funciona para los manejadores definidos como funciones de flecha: las funciones de flecha siempre tienen el mismo valor `this` que el ámbito en el que se definen.

## VALOR DE RETORNO DEL MANIPULADOR

En el JavaScript moderno, los manejadores de eventos no deberían devolver nada. Puedes ver manejadores de eventos que devuelven valores en código antiguo, y el valor de retorno es típicamente una señal al navegador de que no debe realizar la acción por defecto asociada con el evento. Si el manejador `onclick` de un botón de envío en un formulario devuelve falso, por ejemplo, entonces el navegador web no enviará el formulario (generalmente porque el manejador de eventos determinó que la entrada del usuario falla la validación del lado del cliente).

La forma estándar y preferida para evitar que el navegador realice una acción por defecto es llamar al método preventDefault() ([§15.2.5](#)) en el objeto Evento.

## ORDEN DE INVOCACIÓN

Un objetivo de evento puede tener más de un manejador de evento registrado para un tipo de evento en particular. Cuando ocurre un evento de ese tipo, el navegador invoca todos los manejadores en el orden en que fueron registrados. Curiosamente, esto es cierto incluso si se mezclan manejadores de eventos registrados con addEventListener() con un manejador de eventos registrado en una propiedad de objeto como onclick.

### 15.2.4 Propagación de eventos

Cuando el objetivo de un evento es el objeto Window o algún otro objeto independiente, el navegador responde a un evento simplemente invocando los manejadores apropiados en ese objeto. Sin embargo, cuando el objetivo del evento es un documento o un elemento del documento, la situación es más complicada.

Después de invocar los manejadores de eventos registrados en el elemento objetivo, la mayoría de los eventos "burbujean" hacia arriba en el árbol DOM. Se invocan los manejadores de eventos del padre del objetivo. Luego se invocan los manejadores registrados en el abuelo del objetivo. Esto continúa hasta el objeto Documento, y luego más allá del objeto Ventana. El burbujeo de eventos proporciona una alternativa al registro de manejadores en muchos elementos individuales del documento: en su lugar, puedes registrar un único manejador en un elemento ancestro común y manejar los

eventos allí. Puedes registrar un manejador de "cambio" en un elemento <form>, por ejemplo, en lugar de registrar un manejador de "cambio" para cada elemento del formulario.

La mayoría de los eventos que se producen en los elementos del documento burbujean. Las excepciones notables son los eventos "focus", "blur" y "scroll". El evento "load" de los elementos del documento burbujea, pero deja de burbujear en el objeto Documento y no se propaga al objeto Ventana. (Los manejadores de eventos "load" del objeto Window se disparan sólo cuando todo el documento se ha cargado).

El burbujeo de eventos es la tercera "fase" de la propagación de eventos. La invocación de los manejadores de eventos del propio objeto de destino es la segunda fase. La primera fase, que ocurre incluso antes de que se invoquen los manejadores del objeto objetivo, se denomina fase de "captura". Recordemos que addEventListener() toma un tercer argumento opcional. Si ese argumento es true, o {capture:true}, entonces el manejador de eventos es registrado como un manejador de eventos de captura para ser invocado durante esta primera fase de propagación de eventos. La fase de captura de la propagación de eventos es como la fase de burbujeo a la inversa. Los manejadores de captura del objeto Window son invocados primero, luego los manejadores de captura del objeto Document, luego del objeto body, y así sucesivamente hacia abajo en el árbol DOM hasta que los manejadores de captura de eventos del padre del objetivo del evento son invocados. Los manejadores de eventos de captura registrados en el propio objetivo del evento no son invocados.

La captura de eventos proporciona una oportunidad para echar un vistazo a los eventos antes de que sean entregados a su objetivo. Un manejador de eventos de captura puede ser usado para depuración, o puede ser usado junto con la técnica de cancelación de eventos descrita en la siguiente sección para filtrar eventos de manera que los manejadores de eventos de destino nunca sean invocados. Un uso común de la captura de eventos es el manejo de los arrastres del ratón, donde los eventos de movimiento del ratón necesitan ser manejados por el objeto que se arrastra, no por los elementos del documento sobre los que se arrastra.

### **15.2.5 Cancelación de eventos**



Los navegadores responden a muchos eventos del usuario, aunque su código no lo haga: cuando el usuario hace clic con el ratón en un hipervínculo, el navegador sigue el enlace. Si un elemento de entrada de texto HTML tiene el foco del teclado y el usuario escribe una tecla, el navegador introducirá la entrada del usuario. Si el usuario mueve su dedo por un dispositivo de pantalla táctil, el navegador se desplaza. Si registras un manejador de eventos como estos, puedes evitar que el navegador realice su acción por defecto invocando el método `preventDefault()` del objeto de evento. (A menos que hayas registrado el manejador con la opción pasiva, lo que hace que `preventDefault()` no tenga efecto).

La cancelación de la acción por defecto asociada a un evento es sólo un tipo de cancelación de eventos. También podemos cancelar la propagación de eventos llamando al método `stopPropagation()` del objeto evento. Si hay otros manejadores definidos en el mismo objeto, el resto de esos manejadores seguirán siendo invocados, pero no se invocarán manejadores de eventos en ningún otro objeto después de llamar a `stopPropagation()`.

`stopPropagation()` funciona durante la fase de captura, en el propio objetivo del evento y durante la fase de burbujeo.

`stopImmediatePropagation()` funciona como

`stopPropagation()`, pero también impide la invocación de cualquier controlador de eventos posterior registrado en el mismo objeto.

## 15.2.6 Envío de eventos personalizados

La API de eventos del lado del cliente de JavaScript es relativamente potente, y puedes utilizarla para definir y enviar tus propios eventos.

Supongamos, por ejemplo, que tu programa necesita periódicamente realizar un cálculo largo o hacer una petición de red y que, mientras esta operación está pendiente, no es posible realizar otras operaciones. Quieres avisar al usuario de esto mostrando "spinners" para indicar que la aplicación está ocupada. Pero el módulo que está ocupado no necesita saber dónde deben mostrarse los "spinners". En su lugar, ese módulo podría simplemente enviar un evento para anunciar que está ocupado y luego enviar otro evento cuando ya no esté ocupado. Entonces, el módulo de interfaz de usuario puede registrar los manejadores de eventos para esos eventos y tomar cualquier acción de interfaz de usuario que sea apropiada para notificar al usuario.

Si un objeto JavaScript tiene un método `addEventListener()`, entonces es un "objetivo de evento", y esto significa que también tiene un método `dispatchEvent()`. Puedes crear tu propio objeto de evento con el método

`CustomEvent()` y pasarlo a `dispatchEvent()`. El primer argumento de `CustomEvent()` es una cadena que especifica el tipo de su evento, y el segundo argumento es un objeto que especifica las propiedades del objeto del evento. Establezca la propiedad `detail` de este objeto a una cadena, objeto u otro valor que represente el contenido de su evento. Si planeas enviar tu evento sobre un elemento del documento y quieres que se burbuje hacia arriba en el árbol del documento, añade `bubbles:true` al segundo argumento:

```
// Despachar un evento personalizado para que la UI sepa que estamos ocupados
document.dispatchEvent(new CustomEvent("busy", { detail: true }));
```

```
// Realizar una operación de red
```

```
fetch(url) . then(handleNetworkResponse)
  . catch(handleNetworkError)
  . finally(() => {
    // Después de que la petición de red haya tenido éxito o haya fallado, envía // otro
    // evento para que la UI sepa que ya no estamos ocupados.
    document.dispatchEvent(new CustomEvent("busy", { detail: false }));
}

// En otra parte de su programa puede registrar un manejador para
// eventos "ocupados"
// y utilizarlo para mostrar u ocultar el spinner para que el usuario lo sepa.
document.addEventListener("busy", (e) => { if (e.detail) {
  showSpinner(); } else { hideSpinner(); }
});
```

## 15.3 Documentos de scripting

El JavaScript del lado del cliente existe para convertir documentos HTML estáticos en aplicaciones web interactivas. Por lo tanto, el propósito central de JavaScript es crear scripts para el contenido de las páginas web.

Todo objeto Ventana tiene una propiedad Documento que hace referencia a un objeto Documento. El objeto Documento representa el contenido de la ventana, y es el tema de esta sección. Sin embargo, el objeto Document no está solo. Es el objeto central en el DOM para representar y manipular el contenido del documento.

El DOM se introdujo en §15.1.2. Esta sección explica la API en detalle. Abarca:

- Cómo consultar o *seleccionar* elementos individuales de un documento.

Cómo *recorrer* un documento, y cómo encontrar los ancestros, hermanos y descendientes de cualquier elemento del documento.

- Cómo consultar y establecer los atributos de los elementos del documento.
- Cómo consultar, fijar y modificar el contenido de un documento.

Cómo modificar la estructura de un documento creando, insertando y eliminando nodos.

### 15.3.1 Selección de elementos del documento

Los programas JavaScript del lado del cliente a menudo necesitan manipular uno o más elementos dentro del documento. La propiedad global del documento se refiere al objeto Document, y el objeto Document tiene propiedades head y body que se refieren a los objetos Element para las etiquetas <head> y <body>, respectivamente. Pero un programa que quiera manipular un elemento incrustado más profundamente en el documento debe obtener o *seleccionar* de alguna manera los objetos Element que se refieren a esos elementos del documento.

#### SELECCIÓN DE ELEMENTOS CON SELECTORES CSS

Las hojas de estilo CSS tienen una sintaxis muy potente, conocida como *selectores*, para describir elementos o conjuntos de elementos dentro de un documento. Los métodos DOM querySelector() y querySelectorAll() permiten nos permite encontrar el elemento o elementos dentro de un documento que coinciden con un selector CSS especificado. Antes de

cubrir los métodos, comenzaremos con un rápido tutorial sobre la sintaxis del selector CSS.

Los selectores CSS pueden describir elementos por el nombre de la etiqueta, el valor de su atributo id o las palabras de su atributo class:

```
div // Cualquier elemento < div>
#nav // El elemento con id="nav"
.warning // Cualquier elemento con "warning" en su atributo de clase
```

El carácter # se utiliza para buscar en base al atributo id, y el carácter . se utiliza para buscar en base al atributo class. Los elementos también pueden seleccionarse en función de valores de atributos más generales:

```
p[lang="fr"] // Un párrafo escrito en francés: < p lang="fr"> *[name="x"] // Cualquier elemento con un atributo name="x"
```

Tenga en cuenta que estos ejemplos combinan un selector de nombre de etiqueta (o el comodín \* de nombre de etiqueta) con un selector de atributo. También son posibles combinaciones más complejas:

```
span.fatal.error // Cualquier < span> con "fatal" y "error" en su clase
span[lang="fr"].warning // Cualquier < span> en francés con clase "warning"
```

Los selectores también pueden especificar la estructura del documento:

```
#log span // Cualquier < span> descendiente del elemento con id="log"
#log> span // Cualquier < span> hijo del elemento con id="log" body> h1:first-child // El primer < h1> hijo del < body> img + p. caption // Un < p> con clase "caption" inmediatamente después de un < img> h2 ~ p // Cualquier < p> que siga a un < h2> y sea hermano de éste.
```

Si dos selectores están separados por una coma, significa que hemos seleccionado elementos que coinciden con cualquiera de los selectores:

```
button, input[type="button"] // Todos los elementos <button> y <input type="button">.
```

Como puedes ver, los selectores CSS nos permiten referirnos a elementos dentro de un documento por tipo, ID, clase, atributos y posición dentro del documento. El método `querySelector()` toma una cadena de selectores CSS como argumento y devuelve el primer elemento coincidente en el documento que encuentra, o devuelve `null` si no coincide ninguno:

```
// Encuentra el elemento del documento para la etiqueta HTML con el atributo  
// id="spinner"  
let spinner = document.querySelector("#spinner");
```

`querySelectorAll()` es similar, pero devuelve todos los elementos coincidentes del documento en lugar de devolver sólo el primero:

```
// Buscar todos los objetos Element para las etiquetas <h1>, <h2> y <h3>  
let titles =  
document.querySelectorAll("h1, h2, h3");
```

El valor de retorno de `querySelectorAll()` no es un array de Objetos de elemento. En su lugar, es un objeto tipo array conocido como `NodeList`. Los objetos `NodeList` tienen una propiedad de longitud y pueden ser indexados como arrays, por lo que puedes hacer un bucle sobre ellos con un bucle `for` tradicional. Los `NodeLists` también son iterables, por lo que también puedes utilizarlos con bucles `for/of`. Si quieres convertir un `NodeList` en un verdadero array, simplemente pásalo a `Array.from()`.

La NodeList devuelta por querySelectorAll() tendrá una propiedad length establecida en 0 si no hay ningún elemento en el documento que coincida con el selector especificado.

querySelector() y querySelectorAll() están implementados por la clase Element y por la clase Document. Cuando se invocan sobre un elemento, estos métodos sólo devolverán los elementos que sean descendientes de ese elemento.

Tenga en cuenta que CSS define ::first-line y ::first-letter pseudoelementos. En CSS, estos coinciden con porciones de nodos de texto en lugar de elementos reales. No coincidirán si se utilizan con

querySelectorAll() o querySelector(). Además, muchos los navegadores se negarán a devolver coincidencias para las pseudoclases :link y :visited, ya que esto podría exponer información sobre el historial de navegación del usuario.

Otro método de selección de elementos basado en CSS es closest(). Este método está definido por la clase Element y toma un selector como único argumento. Si el selector coincide con el elemento sobre el que se invoca, devuelve ese elemento. En caso contrario, devuelve el elemento antecesor más cercano con el que coincide el selector, o devuelve null si no coincide ninguno. En cierto sentido, closest() es lo contrario de querySelector(): closest() comienza en un elemento y busca una coincidencia por encima de él en el árbol, mientras que querySelector() comienza con un elemento y busca una coincidencia por debajo de él en el árbol. closest() puede ser útil cuando se ha registrado un controlador de eventos en un nivel alto en el árbol del documento. Si está manejando un evento "clic", por

ejemplo, puede querer saber si es un clic un hipervínculo. El objeto de evento le dirá cuál era el objetivo, pero ese objetivo podría ser el texto dentro de un enlace en lugar de la propia etiqueta `<a>` del hipervínculo. Su manejador de eventos podría buscar el hipervínculo contenedor más cercano de esta manera:

```
// Encuentra la etiqueta <a> más cercana que tenga un atributo href. let
hyperlink = event.target.closest("a[href]");
```

Esta es otra forma de utilizar `closest()`:

```
// Devuelve true si el elemento e está dentro de un elemento de lista HTML function
insideList(e) { return e.closest("ul,ol,dl") !== null; }
```

El método relacionado `matches()` no devuelve ancestros o descendientes: simplemente comprueba si un elemento coincide con un selector CSS y devuelve `true` si es así y `false` en caso contrario:

```
// Devuelve true si e es un elemento de encabezamiento HTML
function isHeading(e) { return e.matches("h1,h2,h3,h4,h5,h6"); }
```

## OTROS MÉTODOS DE SELECCIÓN DE ELEMENTOS

Además de `querySelector()` y `querySelectorAll()`, el DOM también define una serie de métodos de selección de elementos más antiguos que ya están más o menos obsoletos. Sin embargo, es posible que aún veas algunos de estos métodos (especialmente `getElementById()`) en uso:

```
// Buscar un elemento por su id. El argumento es sólo el id, sin
// el prefijo del selector CSS #. Similar a
```

```

document.querySelector("#sect1") let sect1 = document.
getElementById("sect1");

// Busca todos los elementos (como las casillas de verificación de los formularios) que
tengan un name="color"
// atributo. Similar a document.querySelectorAll('*[name="color"]'); let colors = document.
getElementsByName("color");

// Buscar todos los elementos <h1> en el documento. // Similar a
document.querySelectorAll("h1") let headings = document.
getElementsByTagName("h1");

// getElementsByTagName() también se define en los elementos.
// Obtener todos los elementos <h2> dentro del elemento sect1. let
subheads = sect1.getElementsByTagName("h2");

// Busca todos los elementos que tienen la clase "tooltip".
// Similar a document.querySelectorAll(".tooltip") let tooltips = document.
getElementsByClassName("tooltip");

// Busca todos los descendientes de sect1 que tengan la clase "sidebar"
// Similar a sect1.querySelectorAll(".sidebar") let sidebars =
sect1.getElementsByClassName("sidebar");

```

Al igual que querySelectorAll(), los métodos de este código devuelven un

NodeList (excepto getElementById(), que devuelve un único objeto Elemento). Sin embargo, a diferencia de querySelectorAll(), el

Las NodeLists devueltas por estos antiguos métodos de selección son "vivas", lo que significa que la longitud y el contenido de la lista pueden cambiar si el contenido o la estructura del documento cambian.

## ELEMENTOS PRESELECCIONADOS

Por razones históricas, la clase Document define propiedades de acceso directo para acceder a ciertos tipos de nodos. Las propiedades de imágenes, formularios y enlaces, por ejemplo,

proporcionan un fácil acceso a los elementos <img>, <form> y <a> (pero sólo a las etiquetas <a> que tienen un atributo href) de un documento. Estas propiedades hacen referencia a objetos HTMLCollection, que son muy parecidos a los objetos NodeList, pero que además pueden ser indexados por el ID o el nombre del elemento. Con la propiedad document.forms, por ejemplo, se puede acceder a la etiqueta <form id="address"> como:

```
documento.formularios.dirección;
```

Una API aún más anticuada para seleccionar elementos es la propiedad document.all, que es como una HTMLCollection para todos los elementos del documento. document.all está obsoleta, y ya no deberías usarla.

### 15.3.2 Estructura de los documentos y su recorrido

Una vez que se ha seleccionado un elemento de un documento, a veces es necesario encontrar partes estructuralmente relacionadas (padres, hermanos, hijos) del documento. Cuando nos interesan principalmente los Elementos de un documento en lugar del texto que contienen (y los espacios en blanco entre ellos, que también son texto), existe una API de navegación que nos permite tratar un documento como un árbol de objetos Elemento, ignorando los nodos de Texto que también forman parte del documento. Esta API de navegación no implica ningún método; es simplemente un conjunto de propiedades en los objetos Elemento que nos permiten referirnos al padre, a los hijos y a los hermanos de un elemento dado:

*parentNode*

Esta propiedad de un elemento se refiere al padre del elemento, que será otro Elemento o un objeto Documento.

### *niños*

Esta NodeList contiene los hijos de un elemento, pero excluye los hijos que no son elementos, como los nodos de texto (y los nodos de comentario).

### *childElementCount*

El número de hijos del elemento. Devuelve el mismo valor que `children.length`.

### *firstElementChild, lastElementChild*

Estas propiedades se refieren al primer y último elemento hijo de un Elemento. Son nulos si el elemento no tiene hijos.

### *nextElementSibling, previousElementSibling*

Estas propiedades se refieren a los elementos hermanos inmediatamente anteriores o posteriores a un elemento, o son nulas si no existe tal hermano.

Utilizando estas propiedades de los elementos, se puede hacer referencia al segundo elemento hijo del primer elemento hijo del documento con cualquiera de estas expresiones:

```
document.children[0].children[1] document.firstElementChild.firstElementChild.nextElementSibling
```

(En un documento HTML estándar, ambas expresiones se refieren al <body> del documento).

Aquí hay dos funciones que demuestran cómo se pueden utilizar estas propiedades para hacer recursivamente un recorrido en profundidad de un documento invocando una función especificada para cada elemento del documento:

```
// Recorrer recursivamente el Documento o Elemento e, invocando la función // f sobre e y sobre cada uno de sus descendientes function traverse(e, f) { f(e); // Invocar f() sobre e for(let child of e.children) { // Iterar sobre los hijos traverse(child, f); // Y recurrir sobre cada uno } }

function traverse2(e, f) { f(e); // Invocar f() sobre e let child = e.firstElementChild; // Iterar el estilo de la lista enlazada de children while(child != null) { traverse2(child, f); // Y recurrir child = child.nextElementSibling; }
}
```

## DOCUMENTOS COMO ÁRBOLES DE NODOS

Si quiere recorrer un documento o alguna parte del mismo y no quiere ignorar los nodos de Texto, puede utilizar un conjunto diferente de propiedades definidas en todos los objetos Nodo. Esto le permitirá ver los Elementos, los nodos de Texto e incluso los nodos de Comentario (que representan los comentarios HTML en el documento).

Todos los objetos Node definen las siguientes propiedades:

### *parentNode*

El nodo que es el padre de éste, o null para nodos como el objeto Documento que no tienen parent.

### *childNodes*

Un NodeList de sólo lectura que contiene todos los hijos (no sólo los hijos del elemento) del nodo.

### *firstChild, lastChild*

El primer y último nodo hijo de un nodo, o null si el nodo no tiene hijos.

*hermanoSiguiente, hermanoAnterior*

Los nodos hermanos siguientes y anteriores de un nodo. Estas propiedades conectan nodos en una lista doblemente enlazada.

*nodeType*

Un número que especifica qué tipo de nodo es. Los nodos de documento tienen el valor 9. Los nodos de elemento tienen el valor 1. Los nodos de texto tienen el valor 3. Los nodos de los comentarios tienen valor 8.

*nodeValue*

El contenido textual de un nodo de Texto o Comentario.

*nodeName*

El nombre de la etiqueta HTML de un elemento, convertido a mayúsculas.

Utilizando estas propiedades del Nodo, se puede hacer referencia al segundo nodo hijo del primer hijo del Documento con expresiones como estas:

`document.childNodes[0].childNodes[1] document.firstChild.firstChild.nextSibling`

Supongamos que el documento en cuestión es el siguiente:

```
<html><head><title> Prueba</title></head><body> ¡Hola mundo!
</body></html>
```

Entonces el segundo hijo del primer hijo es el elemento <body>.

Tiene un nodeType de 1 y un nodeName de "BODY".

Tenga en cuenta, sin embargo, que esta API es extremadamente sensible a las variaciones en el texto del documento. Si el documento se modifica insertando una sola línea nueva entre la etiqueta <html> y la etiqueta <head>, por ejemplo, el nodo Text que representa esa línea nueva se convierte en el primer hijo del primer hijo, y el segundo hijo es el elemento <head> en lugar del elemento <body>.

Para demostrar esta API de recorrido basada en nodos, he aquí una función que devuelve todo el texto dentro de un elemento o documento:

```
// Devuelve el contenido en texto plano del elemento e, recursando en los elementos hijos. // Este método funciona como la propiedad textContent
function textContent(e) {
  let s = ""; // Acumula el texto aquí
  for(let child = e.firstChild; child !== null; child = child.nextSibling) {
    let type = child.nodeType;
    if (type === 3) { // Si es un nodo Text
      s += child.nodeValue; // añade el contenido de texto a nuestra cadena.
    } else if (type === 1) { // Y si es un nodo Elemento
      s += textContent(child); // entonces recurre.
    }
  }
  return s;
}
```

Esta función es sólo una demostración -en la práctica, simplemente escribirías `e.textContent` para obtener el contenido textual del elemento `e`.

### 15.3.3 Atributos

Los elementos HTML constan de un nombre de etiqueta y un conjunto de pares nombre/valor conocidos como *atributos*. El elemento <a> que define un hipervínculo, por ejemplo, utiliza el valor de su atributo href como destino del enlace.

La clase Element define las funciones generales `getAttribute()`, `setAttribute()`, `hasAttribute()` y `removeAttribute()` para consultar, establecer, probar y eliminar los atributos de un elemento. Pero los valores de los atributos de los

elementos HTML (para todos los atributos estándar de los elementos HTML estándar) están disponibles como propiedades de los objetos HTMLElement que representan esos elementos, y normalmente es mucho más fácil trabajar con ellos como propiedades de JavaScript que llamar a `getAttribute()` y métodos relacionados.

## ATRIBUTOS HTML COMO PROPIEDADES DE LOS ELEMENTOS

Los objetos Element que representan los elementos de un documento HTML suelen definir propiedades de lectura/escritura que reflejan los atributos HTML de los elementos. Element define propiedades para los atributos HTML universales como `id`, `title`, `lang` y `dir` y propiedades de manejadores de eventos como `onclick`. Los subtipos específicos de elementos definen atributos específicos de esos elementos. Para consultar la URL de una imagen, por ejemplo, se puede utilizar la propiedad `src` del HTMLElement que representa el elemento `<img>`:

```
let image = document.querySelector("#imagen_principal"); let url = image.src; // El atributo src es la URL de la imagen. id === "imagen_principal" // => true; buscamos la imagen por id
```

Del mismo modo, podrías establecer los atributos de envío de formularios de un elemento `<form>` con un código como este:

```
let f = document.querySelector("form"); // Primer <form> del documento
f.action = "https://www.example.com/submit"; // Establece la URL a la que se enviará.
f.method = "POST"; // Establecer el tipo de petición HTTP.
```

Para algunos elementos, como el elemento `<input>`, algunos nombres de atributos HTML se corresponden con propiedades de nombres diferentes. El atributo HTML `value` de un `<input>`, por ejemplo, se refleja en la propiedad JavaScript `defaultValue`. La

propiedad JavaScript value del elemento <input> contiene la entrada actual del usuario, pero los cambios en la propiedad value no afectan a la propiedad defaultValue ni al atributo value.

Los atributos HTML no distinguen entre mayúsculas y minúsculas, pero los nombres de las propiedades JavaScript sí. Para convertir el nombre de un atributo en una propiedad JavaScript, escríbalo en minúsculas. Sin embargo, si el atributo tiene más de una palabra, ponga la primera letra de cada palabra después de la primera en mayúsculas:

defaultChecked y tabIndex, por ejemplo. Sin embargo, las propiedades de los manejadores de eventos como onclick son una excepción y se escriben en minúsculas.

Algunos nombres de atributos HTML son palabras reservadas en JavaScript. Para ellos, la regla general es anteponer al nombre de la propiedad el prefijo "html". El atributo HTML for (del elemento <label>), por ejemplo, se convierte en la propiedad JavaScript htmlFor. "class" es una palabra reservada en JavaScript, y el importantísimo atributo HTML class es una excepción a la regla: se convierte en className en el código JavaScript.

Las propiedades que representan atributos HTML suelen tener valores de cadena. Pero cuando el atributo es un valor booleano o numérico (los atributos defaultChecked y maxLength de un <input> por ejemplo), las propiedades son booleanos o números en lugar de cadenas. Los atributos de los manejadores de eventos siempre tienen funciones (o null) como valores.

Tenga en cuenta que esta API basada en propiedades para obtener y establecer valores de atributos no define ninguna forma de eliminar un atributo de un elemento. En particular, el operador delete no puede ser utilizado para este propósito. Si necesita eliminar un atributo, utilice el método removeAttribute().

## EL ATRIBUTO DE CLASE

El atributo class de un elemento HTML es especialmente importante. Su valor es una lista separada por espacios de las clases CSS que se aplican al elemento y afectan a su estilo con CSS. Como class es una palabra reservada en JavaScript, el valor de este atributo está disponible a través de la propiedad className en los objetos Element. La propiedad className puede establecer y devolver el valor del atributo class como una cadena. Pero el atributo class está mal nombrado: su valor es una lista de clases CSS, no una sola clase, y es común en la programación JavaScript del lado del cliente querer añadir y eliminar nombres de clases individuales de esta lista en lugar de trabajar con la lista como una sola cadena.

Por esta razón, los objetos Element definen una propiedad classList que permite tratar el atributo class como una lista. El valor de la propiedad classList es un objeto iterable tipo Array. Aunque el nombre de la propiedad es classList, se comporta más como un conjunto de clases, y define los métodos add(), remove(), contains() y toggle():

```
// Cuando queremos que el usuario sepa que estamos ocupados, mostramos
// un spinner. Para ello tenemos que eliminar la clase "hidden" y añadir la clase //
// "animated" (suponiendo que las hojas de estilo estén configuradas correctamente).
let spinner = document.querySelector("#spinner"); spinner.classList.remove("hidden");
spinner.classList.add("animated");
```

## ATRIBUTOS DEL CONJUNTO DE DATOS

A veces es útil adjuntar información adicional a los elementos HTML, normalmente cuando el código JavaScript va a seleccionar esos elementos y manipularlos de alguna manera. En HTML, cualquier atributo cuyo nombre esté en minúsculas y comience con el prefijo "data-" se considera válido, y puedes utilizarlos para cualquier propósito. Estos "atributos de conjunto de datos" no afectarán a la presentación de los elementos en los que aparecen, y definen una forma estándar de adjuntar datos adicionales sin comprometer la validez del documento.

En el DOM, los objetos Element tienen una propiedad dataset que hace referencia a un objeto que tiene propiedades que corresponden a los atributos data- con su prefijo eliminado. Así, dataset.x contendría el valor del atributo data-x. Los atributos con guiones se asignan a nombres de propiedades en camelCase: el atributo data-section-number se convierte en la propiedad dataset.sectionNumber.

Supongamos que un documento HTML contiene este texto:

```
<h2 id="title" data-section-number="16.1"> Atributos</h2>
```

Entonces podrías escribir un JavaScript como este para acceder a ese número de sección:

```
dejar que el número =
document.querySelector("#title").dataset.sectionNumber;
```

### 15.3.4 Contenido de los elementos

Vuelve a mirar el árbol de documentos de la [Figura 15-1](#) y pregúntate cuál es el "contenido" del elemento <p>. Hay dos maneras de responder a esta pregunta:

- El contenido es la cadena HTML "Este es un documento <i>simple</i>".
- El contenido es la cadena de texto plano "Este es un documento simple".

Ambas respuestas son válidas, y cada una de ellas es útil a su manera. Las secciones siguientes explican cómo trabajar con la representación HTML y la representación en texto plano del contenido de un elemento.

## CONTENIDO DEL ELEMENTO COMO HTML

La lectura de la propiedad innerHTML de un elemento devuelve el contenido de ese elemento como una cadena de marcado. Al establecer esta propiedad en un elemento se invoca el analizador del navegador web y se sustituye el contenido actual del elemento por una representación analizada de la nueva cadena. Puedes probar esto abriendo la consola de desarrollador y escribiendo

```
document.body.innerHTML = "<h1>Oops</h1>";
```

Verá que toda la página web desaparece y es sustituida por el único título "Oops". Los navegadores web son muy buenos en el análisis de HTML, y el establecimiento de innerHTML suele ser bastante eficiente. Tenga en cuenta, sin embargo, que añadir texto a la propiedad innerHTML con el operador += no es eficiente porque requiere un paso de serialización para convertir el contenido del

elemento en una cadena y luego un paso de análisis para convertir la nueva cadena de nuevo en contenido del elemento.

### ADVERTENCIA

Cuando se utilizan estas APIs de HTML, es muy importante que nunca se inserte la entrada del usuario en el documento. Si lo haces, permites que usuarios maliciosos injeten sus propios scripts en tu aplicación. Consulte "[Cross-site scripting](#)" para más detalles.

La propiedad outerHTML de un elemento es como innerHTML, salvo que su valor incluye el propio elemento. Cuando se consulta outerHTML, el valor incluye las etiquetas de apertura y cierre del elemento. Y cuando se establece outerHTML en un elemento, el nuevo contenido reemplaza al propio elemento.

Un método relacionado con Element es insertAdjacentHTML(), que permite insertar una cadena de marca HTML arbitraria "adyacente" al elemento especificado. La marca se pasa como segundo argumento a este método, y el significado preciso de "adyacente" depende del valor del primer argumento. Este primer argumento debe ser una cadena con uno de los valores "beforebegin", "afterbegin", "beforeend" o "afterend". Estos valores corresponden a los puntos de inserción que se ilustran en la [Figura 15-2](#).

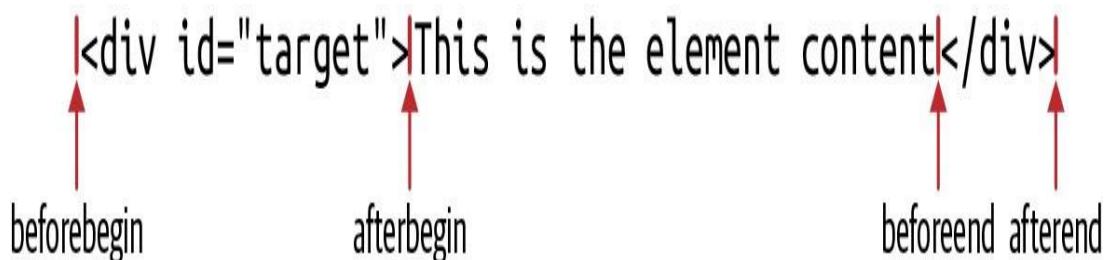


Figura 15-2. Puntos de inserción para insertAdjacentHTML()

## CONTENIDO DEL ELEMENTO COMO TEXTO PLANO

A veces se quiere consultar el contenido de un elemento como texto plano o insertar texto plano en un documento (sin tener que escapar los paréntesis angulares y los ampersands utilizados en el marcado HTML). La forma estándar de hacerlo es con la propiedad `textContent`:

```
let para = document.querySelector("p"); // Primer <p> del documento
let text = para.textContent; // Obtener el texto del párrafo
para.textContent = "¡Hola Mundo!"; // Alterar el texto del párrafo
```

La propiedad `textContent` está definida por la clase `Node`, por lo que funciona tanto para nodos `Text` como para nodos `Element`. Para los nodos `Element`, encuentra y devuelve todo el texto en todos los descendientes del elemento.

La clase `Element` define una propiedad `innerText` que es similar a `textContent`. `innerText` tiene algunas características inusuales y complejas

comportamientos, como el intento de preservar el formato de las tablas. Sin embargo, no está bien especificado ni implementado de forma compatible entre navegadores, por lo que no debería seguir utilizándose.

### TEXTO EN ELEMENTOS <SCRIPT>

Los elementos `<script>` en línea (es decir, los que no tienen un atributo `src`) tienen una propiedad `text` que puedes utilizar para recuperar su texto. El contenido de un elemento `<script>` nunca es mostrado por el navegador, y el analizador HTML ignora los paréntesis angulares y los ampersands dentro de un script. Esto hace que un elemento `<script>` sea un lugar ideal para incrustar datos textuales arbitrarios para su uso en la aplicación. Simplemente establezca el atributo `type` del elemento a algún valor (como `"text/x-custom-data"`) que deje claro que el script no es código JavaScript ejecutable. Si haces esto, el intérprete de JavaScript ignorará el script, pero el elemento existirá en el árbol del documento, y su propiedad `text` te devolverá los datos.

### 15.3.5 Crear, insertar y eliminar nodos

Hemos visto cómo consultar y alterar el contenido del documento utilizando cadenas de

HTML y de texto plano. Y también hemos visto que podemos atravesar un Documento para examinar los nodos individuales de Elemento y Texto que lo componen. También es posible alterar un documento a nivel de nodos individuales. La clase Document define métodos para crear objetos Element, y los objetos Element y Text tienen métodos para insertar, borrar y reemplazar nodos en el árbol.

Crear un nuevo elemento con el método createElement() de la clase Document y añadirle cadenas de texto u otros elementos con sus métodos append() y prepend():

```
let paragraph = document.createElement("p"); // Crear un elemento <p> vacío
let emphasis = document.createElement("em"); // Crear un elemento <em> vacío
emphasis.append("World"); // Añadir texto al elemento <em>.
paragraph.append("¡Hola ", énfasis, "!"); // Añadir texto y <em> a <p> paragraph.
prepend("¡"); // Añadir más texto al principio de <p>
párrafo.innerHTML //=> "Hola
<em>¡Mundo</em>"
```

append() y prepend() toman cualquier número de argumentos, que pueden ser objetos Node o cadenas. Los argumentos de cadena se convierten automáticamente en nodos de texto. (Se pueden crear nodos de texto explícitamente con document.createTextNode(), pero rara vez hay razón para hacerlo). append() añade los argumentos al elemento al final de la lista de hijos. prepend() añade los argumentos al principio de la lista de hijos.

Si quiere insertar un nodo Elemento o Texto en medio de la lista de hijos del elemento contenedor, entonces ni append() ni prepend() le funcionarán. En este caso, debe obtener una referencia a un nodo hermano y llamar a before() para insertar el nuevo contenido antes de ese hermano o after() para insertarlo después de ese hermano.

Por ejemplo:

```
// Buscar el elemento de encabezamiento con class="saludos" let saludos =  
document.querySelector("h2.saludos");  
  
// Ahora inserta el nuevo párrafo y una regla horizontal después de ese encabezado  
saludos.after(párrafo, document.createElement("hr"));
```

Al igual que append() y prepend(), after() y before() toman cualquier número de argumentos de cadena y de elemento y los insertan todos en el documento después de convertir las cadenas en nodos de texto. append() y prepend() sólo se definen en objetos de elemento, pero after() y before() funcionan tanto en nodos de elemento como de texto: puede utilizarlos para insertar contenido relativo a un nodo de texto.

Tenga en cuenta que los elementos sólo pueden insertarse en un punto del documento. Si un elemento ya está en el documento y lo inserta en otro lugar, se moverá a la nueva ubicación, no se copiará:

```
// Hemos insertado el párrafo después de este elemento, pero ahora  
// moverlo para que aparezca antes del elemento en lugar de saludos. before(párrafo);
```

Si quieras hacer una copia de un elemento, utiliza el método cloneNode(), pasando true para copiar todo su contenido:

```
// Haga una copia del párrafo e insértelo después del elemento de saludo  
saludos.after(párrafo.cloneNode(true));
```

Puedes eliminar un nodo Elemento o Texto del documento llamando a su método `remove()`, o puedes reemplazarlo llamando a `replaceWith()` en su lugar. `remove()` no toma argumentos, y `replaceWith()` toma cualquier número de cadenas y elementos al igual que `before()` y `after()`:

```
// Elimina el elemento saludos del documento y lo sustituye por  
// el elemento párrafo (moviendo el párrafo de su ubicación actual // si ya está  
insertado en el documento). greetings.replaceWith(paragraph); // Y ahora  
elimina el párrafo.
```

```
párrafo.remove();
```

La API del DOM también define una generación anterior de métodos para insertar y eliminar contenido. `appendChild()`, `insertBefore()`, `replaceChild()` y `removeChild()` son más difíciles de usar que los métodos mostrados aquí y nunca deberían ser necesarios.

### 15.3.6 Ejemplo: Generación de un índice de contenidos

El ejemplo 15-1 muestra cómo crear dinámicamente una tabla de contenidos para un documento. Demuestra muchas de las técnicas de scripting de documentos descritas en las secciones anteriores. El ejemplo está bien comentado, y no debería tener problemas para seguir el código.

*Ejemplo 15-1. Generación de una tabla de contenidos con la API DOM*

---

```
/** * TOC.js: crea una tabla de contenidos para un documento.  
*  
*      Este script se ejecuta cuando se dispara el evento DOMContentLoaded y genera  
automáticamente una tabla de contenidos para el documento.  
*      No define ningún símbolo global por lo que no debería entrar en conflicto  
*      con otros guiones.  
  
*      Cuando este script se ejecuta, primero busca un elemento del documento con  
*      un id de "TOC". Si no existe tal elemento, crea uno en el  
*      del documento. A continuación, la función encuentra todos los <h2> hasta  
*      <h6>, las trata como títulos de sección y crea una tabla de  
*      contenido dentro del elemento TOC. La función añade números de sección  
*      a cada título de la sección y envuelve los títulos en anclas con nombre para que
```

```
*      para que el TOC pueda enlazarlos. Los anclajes generados tienen nombres
*      que comienzan con "TOC", por lo que debe evitar este prefijo en su propio * HTML.
*
*      Las entradas de la TOC generada pueden ser estilizadas con CSS. Todos los
*      las entradas tienen una clase "TOCEntry". Las entradas también tienen una clase
*      corresponde al nivel del título de la sección. Las etiquetas <h1> generan
*      entradas de la clase "TOCLevel1", las etiquetas <h2> generan entradas de la clase
*      "TOCLevel2", y así sucesivamente. Los números de sección insertados en los títulos
tienen
*      clase "TOCSectNum".
*
*      * Puede utilizar este script con una hoja de estilo como esta:
*      * #TOC { borde: negro sólido 1px; margen: 10px; padding:
10px; }
*          .TOCEntry { margen: 5px 0px; }
*          .TOCEntry a { text-decoration: none; }
*          .TOCLevel1 { font-size: 16pt; font-weight: bold; }
*          .TOCLevel2 { font-size: 14pt; margin-left: .25in; }
*          .TOCLevel3 { font-size: 12pt; margin-left: .5in; }
*          .TOCSectNum:after { contenido: ":"; }
*
*      * Para ocultar los números de sección, utilice esto:
*
*          .TOCSectNum { display: none }
*/
document.addEventListener("DOMContentLoaded", () => {
    // Encuentra el elemento contenedor de la TOC.
    // Si no hay ninguno, crea uno al principio del documento.
    let toc = document.querySelector("#TOC");
    if (! toc) {
        toc = document.createElement("div");
        toc.id = "TOC";
        document.body.prepend(toc);
    }

    // Encontrar todos los elementos de encabezamiento de la sección. Asumimos aquí que el
    // el título del documento utiliza <h1> y que las secciones dentro del
```

```
document están // marcados con <h2> hasta <h6>. let headings = document.  
querySelectorAll("h2,h3,h4,h5,h6");  
  
// Inicializar un array que lleve la cuenta de los números de sección.  
let NúmerosDeSección = [0,0,0,0,0];  
  
// Ahora recorre en bucle los elementos de la cabecera de la sección que hemos  
encontrado.  
for(let heading of headings) { // Omitir el título si está dentro del contenedor TOC. if  
(heading. parentNode === toc) { continue; }  
  
// Averigua de qué nivel es la rúbrica.  
// Restar 1 porque <h2> es un título de nivel 1. let level = parseInt(heading.  
tagName. charAt(1)) - 1;  
  
// Incrementa el número de sección para este nivel de encabezamiento  
// y poner a cero todos los números del nivel inferior de la rúbrica.  
sectionNumbers[level-1]++; for(let i = level; i < sectionNumbers. length; i++) {  
sectionNumbers[i] = 0; }  
  
// Ahora combina los números de sección para todos los niveles de encabezamiento  
// para producir un número de sección como 2.3.1. let sectionNumber =  
sectionNumbers. slice(0, level). join(".");  
  
// Añadir el número de la sección al título de la cabecera de la sección.  
// Colocamos el número en un <span> para que tenga estilo.  
let span = document. createElement("span"); span. className =  
"TOCSectNum"; span. textContent = sectionNumber; heading. prepend(span);  
  
// Envuelve el título en un ancla con nombre para que podamos enlazarlo.  
let anchor = document. createElement("a"); let fragmentName =  
'TOC${sectionNumber}'; anchor. name = fragmentName;
```

```

heading.before(anchor); // Insertar el ancla antes del título anchor.
append(heading); // y mover el título dentro del ancla

// Ahora crea un enlace a esta sección. let link = document.createElement("a"); link.
href = `${fragmentName}`; // Destino del enlace

// Copiar el texto de la cabecera en el enlace. Este es un uso seguro de // innerHTML
porque no estamos insertando ninguna cadena no confiable. link.innerHTML = heading.
innerHTML;

// Coloca el enlace en un div que sea estilizable en función del nivel.
let entry = document.createElement("div"); entry.classList.add("TOCEntry",
`TOCLevel${level}`); entry.append(link);

// Y añade el div al contenedor TOC. toc.append(entry); }

});

```

## 15.4 Scripting CSS

Hemos visto que JavaScript puede controlar la estructura lógica y el contenido de los documentos HTML. También puede controlar la apariencia visual y el diseño de esos documentos mediante el scripting de CSS. Las siguientes subsecciones explican algunas técnicas diferentes que el código JavaScript puede utilizar para trabajar con CSS.

Este es un libro sobre JavaScript, no sobre CSS, y esta sección asume que ya tienes un conocimiento práctico de cómo se usa CSS para dar estilo al contenido HTML. Pero vale la pena mencionar algunos de los estilos CSS que comúnmente se guían desde JavaScript:

- Al establecer el estilo de visualización como "ninguno" se oculta un elemento. Más tarde puede mostrar el elemento estableciendo display a algún otro valor.
- Puede posicionar dinámicamente los elementos estableciendo el estilo de posición como "absoluto", "relativo" o "fijo" y, a continuación, estableciendo los estilos superior e izquierdo en las coordenadas deseadas. Esto es importante cuando se utiliza JavaScript para mostrar contenidos dinámicos como diálogos modales y descripciones de herramientas.
- Puede desplazar, escalar y rotar elementos con el estilo de transformación.
- Puedes animar los cambios de otros estilos CSS con el estilo de transición. Estas animaciones son manejadas automáticamente por el navegador web y no requieren JavaScript, pero puedes usar JavaScript para iniciar las animaciones.

#### 15.4.1 Clases CSS

La forma más sencilla de utilizar JavaScript para afectar al estilo del contenido del documento es añadir y eliminar nombres de clases CSS del atributo class de las etiquetas HTML. Esto es fácil de hacer con la propiedad classList de los objetos Element, como se explica en ["El atributo class"](#).

Suponga, por ejemplo, que la hoja de estilo de su documento incluye una definición para una clase "oculta":

```
.hidden { display:none;  
}
```

Con este estilo definido, puedes ocultar (y luego mostrar) un elemento con un código como este:

```
// Supongamos que este elemento "tooltip" tiene class="hidden" en el archivo HTML.  
// Podemos hacerlo visible así:  
document.querySelector("#tooltip").classList.remove("hidden")  
  
// Y podemos volver a ocultarlo así:  
document.querySelector("#tooltip").classList.add("hidden");  
;
```

### 15.4.2 Estilos en línea

Para continuar con el ejemplo anterior de información sobre herramientas, supongamos que el documento está estructurado con un solo elemento de información sobre herramientas, y queremos posicionarlo dinámicamente antes de mostrarlo. En general, no podemos crear una clase de hoja de estilo diferente para cada posición posible de la información sobre herramientas, por lo que la propiedad `classList` no nos ayudará con el posicionamiento.

En este caso, necesitamos programar el atributo `style` del elemento `tooltip` para establecer estilos en línea que sean específicos de ese elemento. El DOM define una propiedad de estilo en todos los objetos `Element` que se corresponde con el atributo `style`. Sin embargo, a diferencia de la mayoría de estas propiedades, la propiedad `style` no es una cadena. En su lugar, es un objeto `CSSStyleDeclaration`: una representación analizada de los estilos CSS que aparecen en forma textual en el atributo `style`. Para mostrar y establecer la posición de nuestro hipotético tooltip con JavaScript, podríamos utilizar un código como el siguiente

```
function displayAt(tooltip, x, y) { tooltip.style.display = "block"; tooltip.style.position = "absolute"; tooltip.style.left = `${x}px`; tooltip.style.top = `${y}px`; }
```

## CONVENCIONES DE NOMENCLATURA: PROPIEDADES CSS EN JAVASCRIPT

Muchas propiedades de estilo CSS, como font-size, contienen guiones en sus nombres. En JavaScript, un guión se interpreta como un signo menos y no está permitido en los nombres de las propiedades u otros identificadores. Por lo tanto, los nombres de las propiedades del objeto CSSStyleDeclaration son ligeramente diferentes de los nombres de las propiedades CSS reales. Si el nombre de una propiedad CSS contiene uno o más guiones, el nombre de la propiedad CSSStyleDeclaration se forma eliminando los guiones y poniendo en mayúscula la letra que sigue inmediatamente a cada guión. La propiedad CSS border-left-width se accede a través de la propiedad JavaScript borderLeftWidth, por ejemplo, y la propiedad CSS font-family se escribe como fontFamily en JavaScript.

Cuando trabaje con las propiedades de estilo del objeto CSSStyleDeclaration, recuerde que todos los valores deben especificarse como cadenas. En una hoja de estilo o atributo de estilo, puedes escribir:

```
display: block; font-family: sans-serif; background-color: #ffffff;
```

Para lograr lo mismo para un elemento e con JavaScript, hay que citar todos los valores:

```
e. style. display = "block";  
e. style. fontFamily = "sans-serif";  
e. style. backgroundColor = "#ffffff";
```

Tenga en cuenta que los puntos y comas van fuera de las cadenas. Se trata de puntos y comas normales de JavaScript; los puntos y comas que se utilizan en las hojas de estilo CSS no son necesarios como parte de los valores de las cadenas que se establecen con JavaScript.

Además, recuerde que muchas propiedades CSS requieren unidades como "px" para píxeles o "pt" para puntos. Por lo tanto, no es correcto establecer la propiedad marginLeft así:

```
e. style.marginLeft = 300; // Incorrecto: es un número, no una cadena  
e. style.marginLeft = "300"; // Incorrecto: faltan las unidades
```

Las unidades son necesarias cuando se establecen propiedades de estilo en JavaScript, al igual que cuando se establecen propiedades de estilo en las hojas de estilo. La forma correcta de establecer el valor de la propiedad marginLeft de un elemento e a 300 píxeles es:

```
e. style.marginLeft = "300px";
```

Si quieras establecer una propiedad CSS a un valor calculado, asegúrate de añadir las unidades al final del cálculo:

```
e. style.left = `${x0 + left_border + left_padding}px`;
```

Recuerde que algunas propiedades CSS, como margin, son atajos para otras propiedades, como margin-top, margin-right, margin-bottom y margin-left. El objeto CSSStyleDeclaration tiene propiedades que corresponden a estas propiedades de acceso directo. Por ejemplo, puedes establecer la propiedad margin de esta manera:

```
e. style.margin = `${top}px ${right}px ${bottom}px ${left}px`;
```

A veces, puede resultar más fácil establecer o consultar el estilo en línea de un elemento como un único valor de cadena en lugar de como un objeto CSSStyleDeclaration. Para ello, puede utilizar los métodos getAttribute() y setAttribute() de Element, o puede utilizar la propiedad cssText del objeto CSSStyleDeclaration:

// Copiar los estilos inline del elemento e al elemento f:

```
f. setAttribute("style", e.getAttribute("style"));
```

// O hazlo así:

```
f.style.cssText = e.style.cssText;
```

Cuando consulte la propiedad `style` de un elemento, tenga en cuenta que sólo representa los estilos en línea de un elemento y que la mayoría de los estilos para la mayoría de los elementos se especifican en hojas de estilo en lugar de en línea.

Además, los valores que se obtienen al consultar la propiedad de estilo utilizarán las unidades y el formato de la propiedad de acceso directo que se utilicen en el atributo HTML, y es posible que el código tenga que realizar un análisis sintáctico sofisticado para interpretarlos. En general, si quieras consultar los estilos de un elemento, probablemente quieras el *estilo computado*, que se discute a continuación.

### 15.4.3 Estilos computados

El estilo computado para un elemento es el conjunto de valores de propiedades que el navegador deriva (o computa) a partir del estilo en línea del elemento más todas las reglas de estilo aplicables en todas las hojas de estilo: es el conjunto de propiedades realmente utilizadas para mostrar el elemento. Al igual que los estilos en línea, los estilos calculados se representan con un objeto `CSSStyleDeclaration`. Sin embargo, a diferencia de los estilos en línea, los estilos computados son de sólo lectura. No puedes establecer estos estilos, pero el objeto `CSSStyleDeclaration` computado para un elemento te permite determinar qué valores de propiedades de estilo utilizó el navegador al renderizar ese elemento.

Obtenga el estilo calculado para un elemento con el método `getComputedStyle()` del objeto `Window`. El primer argumento de este método es el elemento cuyo estilo calculado se desea

obtener. El segundo argumento opcional se utiliza para especificar un pseudoelemento CSS, como "::before" o "::after":

```
let title = document.querySelector("#section1title"); let styles = window.getComputedStyle(title); let beforeStyles = window.getComputedStyle(title, "::before");
```

El valor de retorno de `getComputedStyle()` es un Objeto `CSSStyleDeclaration` que representa todos los estilos que se aplican al elemento (o pseudoelemento) especificado. Hay una serie de diferencias importantes entre un objeto `CSSStyleDeclaration` que representa estilos en línea y uno que representa estilos computados:

- Las propiedades de estilo calculadas son de sólo lectura.
- Las propiedades de estilo calculadas son *absolutas*: las unidades relativas como los porcentajes y los puntos se convierten en valores absolutos. Cualquier propiedad que especifique un tamaño (como un tamaño de margen o un tamaño de fuente) tendrá un valor medido en píxeles. Este valor será una cadena con un sufijo "px", por lo que todavía tendrá que analizarlo, pero no tendrá que preocuparse de analizar o convertir otras unidades. Las propiedades cuyos valores sean colores se devolverán en formato "rgb()" o "rgba()".
- Las propiedades de acceso directo no se calculan, sólo las propiedades fundamentales en las que se basan. No consulte la propiedad `margin`, por ejemplo, sino que utilice `marginLeft`, `marginTop`, etc. Del mismo modo, no consulte el borde o incluso `borderWidth`. En su lugar, utilice `borderLeftWidth`, `borderTopWidth`, etc.
- La propiedad `cssText` del estilo calculado no está definida.

Un objeto CSSStyleDeclaration devuelto por getComputedStyle() generalmente contiene mucha más información sobre un elemento que la CSSStyleDeclaration obtenida de la propiedad inline style de ese elemento. Pero los estilos computados pueden ser complicados, y su consulta no siempre proporciona la información que se espera. Considere el atributo font-family: acepta una lista separada por comas de las familias de fuentes deseadas para la portabilidad entre plataformas. Cuando se consulta la propiedad fontFamily de un estilo calculado, simplemente se obtiene el valor del estilo font-family más específico que se aplica al elemento. Esto puede devolver un valor como "arial,helvetica,sans-serif", que no le dice qué tipo de letra está realmente en uso. Del mismo modo, si un elemento no está absolutamente posicionado, el intento de consultar su posición y tamaño a través de las propiedades top y left de su estilo computado suele devolver el valor auto. Este es un valor CSS perfectamente legal, pero probablemente no es lo que estabas buscando.

Aunque CSS puede utilizarse para especificar con precisión la posición y el tamaño de los elementos del documento, consultar el estilo calculado de un elemento no es la forma preferida de determinar el tamaño y la posición del elemento. Véase en [§15.5.2](#) una alternativa más sencilla y portable.

#### 15.4.4 Hojas de estilo de scripting

Además de los atributos de clase y los estilos en línea, JavaScript también puede manipular las hojas de estilo. Las hojas de estilo se asocian a un documento HTML con una etiqueta <style> o con una etiqueta <link rel="stylesheet">. Ambas son etiquetas HTML

normales, por lo que puedes darles ambos atributos id y luego buscarlas con document.querySelector().

Los objetos Element para las etiquetas <style> y <link> tienen una propiedad disabled que puedes usar para desactivar toda la hoja de estilos. Podrías usarlo con un código como este:

```
// Esta función cambia entre los temas "claro" y "oscuro"
function toggleTheme() {
  let lightTheme = document.querySelector("#luz-tema");
  let darkTheme = document.querySelector("#oscuro-tema");
  if (darkTheme.disabled) { // Actualmente claro, cambia a oscuro
    lightTheme.disabled = true;
    darkTheme.disabled = false;
  } else { // Actualmente oscuro, cambiar a claro
    lightTheme.disabled = false;
    darkTheme.disabled = true;
  }
}
```

Otra forma sencilla de escribir hojas de estilo es insertar otras nuevas en el documento utilizando las técnicas de manipulación del DOM que ya hemos visto.

Por ejemplo:

```
function setTheme(name) {
  // Crear un nuevo elemento <link rel="stylesheet"> para cargar la hoja de estilos
  // nombrada
  let link = document.createElement("link");
  link.id = "theme";
  link.rel = "stylesheet";
  link.href = `themes/${name}.css`;

  // Buscar un enlace existente con el id "tema"
  let existingLink = document.querySelector("#tema");
  if (existingLink) {
    document.head.removeChild(existingLink);
  }
  document.head.appendChild(link);
}
```

```
let currentTheme = document.querySelector("#theme"); if (currentTheme) {  
    // Si hay un tema existente, sustitúyalo por el nuevo.  
    currentTheme.replaceWith(link);  
} si no {  
    // De lo contrario, simplemente inserte el enlace a la hoja de estilos del tema.  
    document.head.append(link);  
}
```

De forma menos sutil, también puedes simplemente insertar una cadena de HTML que contenga un `<style>` en su documento. Este es un truco divertido, por ejemplo:

```
document.head.insertAdjacentHTML(  
    "beforeend",  
    "<style>body{transform:rotate(180deg)}</style>"  
)
```

Los navegadores definen una API que permite a JavaScript mirar dentro de las hojas de estilo para consultar, modificar, insertar y eliminar reglas de estilo en esa hoja de estilo. Esta API es tan especializada que no está documentada aquí. Puedes leer sobre ella en MDN buscando "CSSStyleSheet" y "Modelo de objetos CSS".

#### 15.4.5 Animaciones y eventos CSS

Suponga que tiene las siguientes dos clases CSS definidas en una hoja de estilos:

```
.transparent { opacidad: 0; }  
.fadeable { transición: opacidad .5s ease-in }
```

Si aplicas el primer estilo a un elemento, éste será totalmente transparente y por tanto invisible. Pero si aplicas el segundo estilo que le dice al navegador que cuando la opacidad del elemento cambia, ese cambio debe ser animado durante un período de 0,5 segundos, "ease-in" especifica que la animación del cambio de opacidad debe comenzar lentamente y luego acelerarse.

Ahora supongamos que su documento HTML contiene un elemento con la clase "fadeable":

```
<div id="subscribe" class="fadeable notification">... </div>
```

En JavaScript, puede añadir la clase "transparente":

```
document.querySelector("#subscribe").classList.add("transparent");
```

Este elemento está configurado para animar los cambios de opacidad. Al añadir la clase "transparent" se cambia la opacidad y se desencadena una animación: el navegador "desvanece" el elemento para que se vuelva totalmente transparente durante el periodo de medio segundo.

Esto también funciona a la inversa: si se elimina la clase "transparente" de un elemento "desvanecible", también se produce un cambio de opacidad, y el elemento se desvanece y vuelve a ser visible.

JavaScript no tiene que hacer ningún trabajo para que se produzcan estas animaciones: son un efecto CSS puro. Pero se puede utilizar JavaScript para desencadenarlas.

También se puede utilizar JavaScript para controlar el progreso de una transición CSS, ya que el navegador web lanza eventos al inicio y al final de una transición. El evento "transitionrun" se despacha cuando la transición se dispara por primera vez. Esto puede ocurrir antes de que comiencen los cambios visuales, cuando se ha especificado el estilo de retardo de la transición. Una vez que los cambios visuales comienzan, se envía un evento "transitionstart", y cuando la animación se completa, se envía un evento "transitionend". El objetivo de todos estos eventos es el elemento que se está animando, por supuesto. El objeto de evento que se pasa a los manejadores de estos eventos es un objeto TransitionEvent. Tiene una propiedad `propertyName` que especifica la propiedad CSS que se está animando y una propiedad `elapsedTime` que para los eventos "transitionend" especifica cuántos segundos han pasado desde el evento "transitionstart".

Además de las transiciones, CSS también soporta una forma más compleja de animación conocida simplemente como "Animaciones CSS". Estas utilizan propiedades CSS como `animation-name` y `animation-duration` y una regla especial `@keyframes` para definir los detalles de la animación. Los detalles de cómo funcionan las animaciones CSS están más allá del alcance de este libro, pero una vez más, si defines todas las propiedades de animación en una clase CSS, entonces puedes usar JavaScript para activar la animación simplemente añadiendo la clase al elemento que va a ser animado.

Y al igual que las transiciones CSS, las animaciones CSS también desencadenan eventos que tu código JavaScript puede escuchar. El evento "animationstart" se envía cuando la animación comienza, y el

evento "animationend" se envía cuando termina. Si la animación se repite más de una vez, se enviará un evento

El evento "animationiteration" se envía después de cada repetición, excepto la última. El objetivo del evento es el elemento animado, y el objeto del evento que se pasa a las funciones manejadoras es un objeto AnimationEvent. Estos eventos incluyen una propiedad animationName que especifica la propiedad animation-name que define la animación y una propiedad elapsedTime que especifica cuántos segundos han pasado desde que comenzó la animación.

## **15.5 Geometría del documento y desplazamiento**

En este capítulo hemos pensado hasta ahora en los documentos como árboles abstractos de elementos y nodos de texto. Sin embargo, cuando un navegador representa un



dentro de una ventana, crea una representación visual del documento en la que cada elemento tiene una posición y un tamaño. A menudo, las aplicaciones web pueden tratar los documentos como árboles de elementos y no tener que pensar nunca en cómo se representan esos elementos en la pantalla. Sin embargo, a veces es necesario determinar la geometría precisa de un elemento. Si, por ejemplo, quieras usar CSS para posicionar dinámicamente un elemento (como un tooltip) junto a algún elemento ordinario posicionado por el navegador, necesitas ser capaz de determinar la ubicación de ese elemento.

Las siguientes subsecciones explican cómo se puede ir y venir entre el *modelo abstracto*, basado en el árbol, de un documento y la *vista geométrica*, basada en coordenadas, del documento tal y como se presenta en una ventana del navegador.

### 15.5.1 Coordenadas del documento y coordenadas de la ventana gráfica

La posición de un elemento del documento se mide en píxeles CSS, con la coordenada *x* aumentando hacia la derecha y la coordenada *y* aumentando a medida que bajamos. Sin embargo, hay dos puntos diferentes que podemos utilizar como origen del sistema de coordenadas: las coordenadas *x* e *y* de un elemento pueden ser relativas a la esquina superior izquierda del documento o relativas a la esquina superior izquierda de la *ventana gráfica* en la que se muestra el documento. En las ventanas y pestañas de nivel superior, la "ventana gráfica" es la parte del navegador que realmente muestra el contenido del documento: excluye el "cromo" del navegador, como los menús, las barras de herramientas y las

pestañas. En el caso de los documentos mostrados en etiquetas <iframe>, es el elemento iframe en el DOM el que define la ventana gráfica para el documento anidado. En cualquier caso, cuando hablamos de la posición de un elemento, debemos tener claro si estamos utilizando coordenadas del documento o coordenadas de la ventana gráfica. (Tenga en cuenta que las coordenadas de la ventana gráfica a veces se denominan "coordenadas de ventana").

Si el documento es más pequeño que la ventana gráfica, o si no se ha desplazado, la esquina superior izquierda del documento está en la esquina superior izquierda de la ventana gráfica y los sistemas de coordenadas del documento y de la ventana gráfica son los mismos. En general, sin embargo, para convertir entre los dos sistemas de coordenadas, debemos sumar o restar los *desplazamientos*. Si un elemento tiene una coordenada y de 200 píxeles en las coordenadas del documento, por ejemplo, y si el usuario se ha desplazado hacia abajo 75 píxeles, entonces ese elemento tiene una coordenada y de 125 píxeles en las coordenadas de la ventana gráfica. Del mismo modo, si un elemento tiene una coordenada x de 400 en las coordenadas de la ventana gráfica después de que el usuario se haya desplazado 200 píxeles horizontalmente, entonces la coordenada x del elemento en las coordenadas del documento es 600.

Si utilizamos el modelo mental de los documentos impresos en papel, es lógico suponer que cada elemento de un documento debe tener una posición única en las coordenadas del documento, independientemente de cuánto haya desplazado el usuario el documento. Esta es una propiedad atractiva de los documentos en papel, y se aplica a los documentos web simples, pero en general, las coordenadas del documento no funcionan realmente en la web. El

problema es que la propiedad de desbordamiento de CSS permite que los elementos de un documento tengan más contenido del que puede mostrar. Los elementos pueden tener sus propias barras de desplazamiento y servir como ventanas para el contenido que contienen. El hecho de que la web permita el desplazamiento de elementos dentro de un documento con desplazamiento significa que simplemente no es posible describir la posición de un elemento dentro del documento utilizando un único punto (x,y).

Dado que las coordenadas del documento no funcionan realmente, el JavaScript del lado del cliente tiende a utilizar las coordenadas de la ventana gráfica. La página

`getBoundingClientRect() y elementFromPoint()`

Los métodos que se describen a continuación utilizan las coordenadas de la ventana gráfica, por ejemplo, y las propiedades `clientX` y `clientY` de los objetos de eventos de ratón y puntero también utilizan este sistema de coordenadas.

Cuando se posiciona explícitamente un elemento mediante CSS `position:fixed`, las propiedades `top` y `left` se interpretan en coordenadas de la ventana gráfica. Si se utiliza `position:relative`, el elemento se posiciona en relación con el lugar en el que estaría si no tuviera la propiedad `position` establecida. Si utiliza `position:absolute`, entonces `top` y `left` son relativos al documento o al elemento posicionado más cercano. Esto significa, por ejemplo, que un elemento posicionado absolutamente dentro de un elemento posicionado relativamente se posiciona en relación con el elemento contenedor, no en relación con el documento global. A veces es muy útil crear un contenedor relativamente posicionado con la parte superior y la izquierda ajustadas a 0 (para que el contenedor se

disponga normalmente) con el fin de establecer un nuevo origen del sistema de coordenadas para los elementos absolutamente posicionados que contiene. Podemos referirnos a este nuevo sistema de coordenadas como "coordenadas del contenedor" para distinguirlo de las coordenadas del documento y de las coordenadas de la ventana gráfica.

#### CSS PIXELS

Si, como yo, tiene la edad suficiente para recordar los monitores de ordenador con resoluciones de  $1024 \times 768$  y los teléfonos con pantalla táctil con resoluciones de  $320 \times 480$ , entonces puede que siga pensando que la palabra "píxel" se refiere a un único "elemento de imagen" en el *hardware*. Los monitores 4K y las pantallas "retina" actuales tienen una resolución tan alta que los píxeles de software se han desvinculado de los píxeles de hardware. Un píxel CSS -y, por tanto, un píxel JavaScript del lado del cliente- puede consistir, de hecho, en varios píxeles de dispositivo. La propiedad devicePixelRatio del objeto Window especifica cuántos píxeles de dispositivo se utilizan para cada píxel de software. Un "dpr" de 2, por ejemplo, significa que cada píxel de software es en realidad una cuadrícula de  $2 \times 2$  píxeles de hardware. El valor de devicePixelRatio depende de la resolución física de su hardware, de la configuración de su sistema operativo y del nivel de zoom de su navegador.

devicePixelRatio no tiene que ser un número entero. Si está utilizando un tamaño de fuente CSS de "12px" y la relación de píxeles del dispositivo es 2,5, entonces el tamaño real de la fuente, en píxeles del dispositivo, es 30. Dado que los valores de píxeles que utilizamos en CSS ya no se corresponden directamente con los píxeles individuales de la pantalla, las coordenadas de los píxeles ya no necesitan ser enteras. Si el devicePixelRatio es 3, entonces una coordenada de 3,33 tiene mucho sentido. Y si el ratio es realmente 2, entonces una coordenada de 3,33 se redondeará a 3,5.

### 15.5.2 Consultar la geometría de un elemento

Puede determinar el tamaño (incluyendo el borde y el relleno CSS, pero no el margen) y la posición (en coordenadas de la ventana gráfica) de un elemento llamando a su método getBoundingClientRect(). No toma argumentos y devuelve un objeto con las propiedades izquierda, derecha, arriba, abajo, ancho y alto. Las propiedades izquierda y superior dan las coordenadas x e y de la esquina superior izquierda del elemento, y las propiedades derecha e inferior dan las coordenadas de la esquina inferior derecha. Las

diferencias entre estos valores son las propiedades de anchura y altura.

Los elementos en bloque, como las imágenes, los párrafos y los elementos `<div>`, son siempre rectangulares cuando el navegador los presenta. Sin embargo, los elementos en línea, como los elementos `<span>`, `<code>` y `<b>`, pueden abarcar varias líneas y, por lo tanto, constar de varios rectángulos. Imagina, por ejemplo, un texto dentro de las etiquetas `<em>` y `</em>` que se muestra de forma que se extiende a lo largo de dos líneas. Sus rectángulos consisten en el final de la primera línea y el principio de la segunda. Si llama a `getBoundingClientRect()` en este elemento, el límite incluiría todo el ancho de ambas líneas. Si desea consultar los rectángulos individuales de los elementos en línea, llame al método `getClientRects()` para obtener un objeto de sólo lectura, tipo matriz, cuyos elementos son objetos rectángulo como los devueltos por `getBoundingClientRect()`.

### 15.5.3 Determinación del elemento en un punto

El método `getBoundingClientRect()` nos permite determinar la posición actual de un elemento en una ventana gráfica. A veces queremos ir en la otra dirección y determinar qué elemento está en un lugar determinado de la ventana gráfica. Puedes determinar esto con el método `elementFromPoint()` del objeto Document. Llama a este método con las coordenadas `x` e `y` de un punto (usando las coordenadas de la ventana gráfica, no las coordenadas del documento: las coordenadas `clientX` y `clientY` de un evento de ratón funcionan, por ejemplo).

`elementFromPoint()` devuelve un objeto `Element` que se encuentra en la posición especificada. El algoritmo de *detección de aciertos* para seleccionar el elemento no se especifica con precisión, pero la intención de este método es que devuelva el elemento más interno (más profundamente anidado) y más alto (atributo CSS `zindex` más alto) en ese punto.

#### 15.5.4 Desplazamiento

El método `scrollTo()` del objeto `Window` toma las coordenadas `x` e `y` de un punto (en coordenadas del documento) y las establece como desplazamientos de la barra de desplazamiento. Es decir, desplaza la ventana para que el punto especificado esté en la esquina superior izquierda de la ventana. Si especifica un punto que está demasiado cerca de la parte inferior o demasiado cerca del borde derecho del documento, el navegador lo moverá tan cerca como sea posible de la esquina superior izquierda, pero no será capaz de llegar hasta allí. El siguiente código desplaza el navegador para que la página más inferior del documento sea visible:

```
// Obtener las alturas del documento y de la ventana gráfica.  
let documentHeight = document.documentElement.offsetHeight; let viewportHeight  
= window.innerHeight; // Y desplázate para que la última "página" se muestre en la  
ventana viewport. scrollTo(0, documentHeight - viewportHeight);
```

El método `scrollBy()` de la ventana es similar a `scrollTo()`, pero sus argumentos son relativos y se añaden a la posición de desplazamiento actual:

```
// Desplazar 50 píxeles hacia abajo cada 500 ms. Tenga en cuenta que no hay manera  
de desactivar esto! setInterval(() => { scrollBy(0,50)}, 500);
```

Si quiere desplazarse suavemente con scrollTo() o scrollBy(), pase un único argumento de objeto en lugar de dos números, así:

```
window.scrollTo({ left: 0,  
    top: documentHeight - viewportHeight, behavior:  
    "smooth" });
```

A menudo, en lugar de desplazarse a una ubicación numérica en un documento, sólo queremos desplazarnos para que un determinado elemento del documento sea visible. Esto se puede hacer con el método scrollIntoView() en el elemento HTML deseado. Este método asegura que el elemento sobre el que se invoca es visible en la ventana gráfica. Por defecto, intenta poner el borde superior del elemento en o cerca de la parte superior de la ventana gráfica. Si se pasa false como único argumento, intenta poner el borde inferior del elemento en la parte inferior de la ventana gráfica. El navegador también desplazará la ventana gráfica horizontalmente según sea necesario para hacer visible el elemento.

También puede pasar un objeto a scrollIntoView(), estableciendo la propiedad behavior: "smooth" para un desplazamiento suave. Puede establecer la propiedad "block" para especificar dónde debe posicionarse el elemento verticalmente y la propiedad "inline" para especificar cómo debe posicionarse horizontalmente si es necesario el desplazamiento horizontal. Los valores legales para ambas propiedades son inicio, fin, más cercano y centro.

## 15.5.5 Tamaño de la ventana gráfica, tamaño del contenido y posición de desplazamiento

Como ya hemos comentado, las ventanas del navegador y otros elementos HTML pueden mostrar contenido que se desplaza. Cuando este es el caso, a veces necesitamos conocer el tamaño de la ventana gráfica, el tamaño del contenido y los desplazamientos del contenido dentro de la ventana gráfica. Esta sección cubre estos detalles.

En el caso de las ventanas del navegador, el tamaño de la ventana gráfica viene dado por el parámetro

propiedades `window.innerWidth` y `window.innerHeight`.

(Las páginas web optimizadas para dispositivos móviles suelen utilizar una etiqueta `<meta name="viewport">` en su `<head>` para establecer el ancho de ventana deseado para la página). El tamaño total del documento es el mismo que el tamaño del elemento `<html>`, `document.documentElement`. Puede llamar a

`getBoundingClientRect()` en

`document.documentElement` para obtener la anchura y la altura del documento, o puede utilizar las propiedades `offsetWidth` y `offsetHeight` de `document.documentElement`. Los desplazamientos del documento dentro de su ventana gráfica están disponibles como `window.scrollX` y `window.scrollY`. Estas son propiedades de sólo lectura, por lo que no puedes establecerlas para desplazar el documento: utiliza `window.scrollTo()` en su lugar.

Las cosas son un poco más complicadas para los elementos. Cada objeto Elemento define los siguientes tres grupos de propiedades:

`offsetWidth` `clientWidth` `scrollWidth` `offsetHeight` `clientHeight`  
`scrollHeight` `offsetLeft` `clientLeft` `scrollLeft` `offsetTop` `clientTop`  
`scrollTop` `offsetParent`

Las propiedades `offsetWidth` y `offsetHeight` de un elemento

devuelve su tamaño en pantalla en píxeles CSS. Los tamaños devueltos incluyen el borde y el relleno del elemento, pero no los márgenes. Las propiedades `offsetLeft` y `offsetTop` devuelven las coordenadas `x` e `y` del elemento. Para muchos elementos, estos valores son coordenadas del documento. Pero para los descendientes de los elementos posicionados y para algunos otros elementos, como las celdas de las tablas, estas propiedades devuelven coordenadas que son relativas a un elemento antecesor y no al propio documento. La propiedad `offsetParent` especifica con qué elemento son relativas las propiedades. Todas estas propiedades de desplazamiento son de sólo lectura.

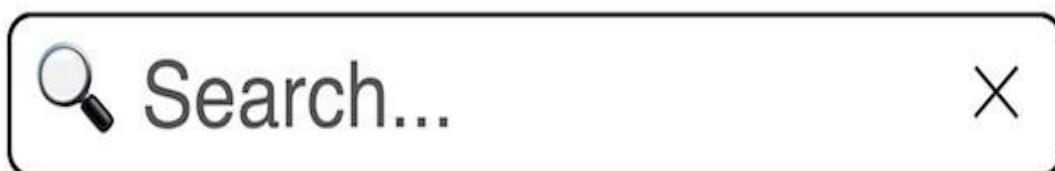
`clientWidth` y `clientHeight` son como `offsetWidth` y `offsetHeight`, excepto que no incluyen el tamaño del borde, sólo el área de contenido y su relleno. Las propiedades `clientLeft` y `clientTop` no son muy útiles: devuelven la distancia horizontal y vertical entre el exterior del relleno de un elemento y el exterior de su borde. Normalmente, estos valores son sólo el ancho de los bordes izquierdo y superior. Todas estas propiedades del cliente son de sólo lectura. Para los elementos en línea como `<i>`, `<code>` y `<span>`, todos devuelven 0.

`scrollWidth` y `scrollHeight` devuelven el tamaño del área de contenido de un elemento más su relleno más cualquier contenido que desborde. Cuando el contenido cabe en el área de contenido sin desbordamiento, estas propiedades son las mismas que `clientWidth` y `clientHeight`. Pero cuando hay desbordamiento, incluyen el contenido desbordado y devuelven los valores mayores que `clientWidth` y `clientHeight`. `scrollLeft` y `scrollTop` dan el desplazamiento del contenido del elemento dentro de la ventana

gráfica del elemento. A diferencia de las otras propiedades descritas aquí, scrollLeft y scrollTop son propiedades escribibles, y puedes establecerlas para desplazar el contenido dentro de un elemento. (En la mayoría de los navegadores, los objetos Element también tienen métodos scrollTo() y scrollBy() como lo hace el objeto Window, pero estos no son todavía universalmente soportados).

## 15.6 Componentes web

HTML es un lenguaje para el marcado de documentos y define un rico conjunto de etiquetas para ese fin. En las últimas tres décadas, se ha convertido en un lenguaje que se utiliza para describir las interfaces de usuario de las aplicaciones web, pero las etiquetas HTML básicas como <input> y <botón> son inadecuadas para los diseños de interfaz de usuario modernos. Los desarrolladores web son capaces de hacerlo funcionar, pero sólo mediante el uso de CSS y JavaScript para aumentar la apariencia y el comportamiento de las etiquetas HTML básicas. Considere un componente típico de la interfaz de usuario, como el cuadro de búsqueda que se muestra en la Figura 15-3.



*Figura 15-3. Un componente de la interfaz de usuario de la caja de búsqueda*

El elemento HTML <input> puede utilizarse para aceptar una sola línea de entrada del usuario, pero no tiene ninguna forma de mostrar iconos como la lupa a la izquierda y la X de cancelación a la derecha. Para implementar un elemento de interfaz de usuario moderno como éste para la web, necesitamos utilizar al menos

cuatro elementos HTML: un elemento `<input>` para aceptar y mostrar la entrada del usuario, dos elementos `<img>` (o en este caso, dos elementos `<span>` que muestren glifos Unicode), y un elemento contenedor `<div>` para contener esos tres hijos. Además, tenemos que utilizar CSS para ocultar el borde por defecto del elemento `<input>` y definir un borde para el contenedor. Y tenemos que usar JavaScript para hacer que todos los elementos HTML funcionen juntos. Cuando el usuario hace clic en el icono X, necesitamos un controlador de eventos para borrar la entrada del elemento `<input>`, por ejemplo.

Eso es mucho trabajo para hacer cada vez que se quiere mostrar un cuadro de búsqueda en una aplicación web, y la mayoría de las aplicaciones web hoy en día no están escritas usando HTML "crudo". En su lugar, muchos desarrolladores web utilizan frameworks como React y Angular que soportan la creación de componentes de interfaz de usuario reutilizables como el cuadro de búsqueda que se muestra aquí. Los componentes web son una alternativa nativa para el navegador a esos marcos basados en tres adiciones relativamente recientes a los estándares web que permiten a JavaScript ampliar el HTML con nuevas etiquetas que funcionan como componentes de interfaz de usuario autónomos y reutilizables.

Las subsecciones que siguen explican cómo utilizar los componentes web definidos por otros desarrolladores en sus propias páginas web, luego explican cada una de las tres tecnologías en las que se basan los componentes web y, finalmente, unen las tres en un ejemplo que implementa el elemento de la caja de búsqueda que se muestra en la [Figura 15-3](#).

## 15.6.1 Uso de los componentes web

Los componentes web se definen en JavaScript, por lo que para utilizar un componente web en su archivo HTML, debe incluir el archivo JavaScript que define el componente. Dado que los componentes web son una tecnología relativamente nueva, a menudo se escriben como módulos de JavaScript, por lo que podría incluir uno en su HTML de la siguiente manera:

```
<script type="module" src="components/search-box.js">
```

Los componentes web definen sus propios nombres de etiquetas HTML, con la importante restricción de que esos nombres de etiquetas deben incluir un guión. (Esto significa que las futuras versiones de HTML pueden introducir nuevas etiquetas sin guiones, y no hay posibilidad de que las etiquetas entren en conflicto con el componente web de nadie). Para utilizar un componente web, basta con usar su etiqueta en el archivo HTML:

```
<búsqueda-box placeholder="Buscar..."></búsqueda-box>
```

Los componentes web pueden tener atributos al igual que las etiquetas HTML normales; la documentación del componente que está utilizando debería indicarle qué atributos son compatibles. Los componentes web no pueden definirse con etiquetas de cierre automático. Por ejemplo, no se puede escribir `<search-box/>`. Su archivo HTML debe incluir tanto la etiqueta de apertura como la de cierre.

Al igual que los elementos regulares de HTML, algunos componentes web están escritos para esperar hijos y otros están escritos de tal manera que no esperan (y no mostrarán) hijos. Algunos

componentes web están escritos de manera que pueden aceptar opcionalmente hijos especialmente etiquetados que aparecerán en "ranuras" con nombre. El componente <búsqueda> que se muestra en la [Figura 15-3](#) y que se implementa en el [Ejemplo 15-3](#) utiliza "ranuras" para los dos iconos que muestra. Si desea utilizar un <search-box> con diferentes iconos, puede utilizar HTML como este:

```
<caja de búsqueda>


</b> <b>Cuadro de búsqueda</b>
```

El atributo slot es una extensión de HTML que se utiliza para especificar qué hijos deben ir a dónde. Los nombres de las ranuras - "izquierda" y "derecha" en este ejemplo- son definidos por el componente web. Si el componente que está utilizando admite ranuras, este hecho debe incluirse en su documentación.

Anteriormente señalé que los componentes web suelen implementarse como módulos de JavaScript y pueden ser cargados en archivos HTML con un <script type="module">. Quizá recuerdes del principio de este capítulo que los módulos se cargan después de analizar el contenido del documento, como si tuvieran una etiqueta diferida. Así que esto significa que un navegador web normalmente analizará y renderizará etiquetas como <search-box> antes de haber ejecutado el código que le dirá qué es un <search-box>. Esto es normal cuando se utilizan componentes web. Los analizadores HTML de los navegadores web son flexibles y muy indulgentes con las entradas que no entienden. Cuando encuentran una etiqueta de componente web antes de que ese componente haya sido definido, añaden un

HTMLElement genérico al árbol DOM aunque no sepan qué hacer con él. Más tarde, cuando se define el elemento personalizado, el elemento genérico se "actualiza" para que se vea y se comporte como se desea.

Si un componente web tiene hijos, es probable que esos hijos se muestren incorrectamente antes de que se defina el componente. Puede utilizar este CSS para mantener ocultos los componentes web hasta que se definan:

```
/*
 *      Hacer invisible el componente <search-box> antes de definirlo.
 *      Y tratar de duplicar su eventual diseño y tamaño para que cerca
 *      El contenido no se mueve cuando se define.
 */ search-box:not(:defined) {
    opacity:0; display: inline-block;
```

```
    ancho: 300px; alto:
    50px; }
```

Al igual que los elementos HTML normales, los componentes web pueden utilizarse en JavaScript. Si incluyes una etiqueta <search-box> en tu página web, puedes obtener una referencia a ella con `querySelector()` y un selector CSS apropiado, igual que harías con cualquier otra etiqueta HTML. Generalmente, sólo tiene sentido hacer esto después de que el módulo que define el componente se haya ejecutado, así que tenga cuidado cuando consulte componentes web de no hacerlo demasiado pronto. Las implementaciones de componentes web suelen (aunque no es un requisito) definir una propiedad JavaScript para cada atributo HTML que soportan. Y, al igual que los elementos HTML, también pueden definir métodos útiles. Una vez más, la documentación del

componente web que está utilizando debería especificar qué propiedades y métodos están disponibles para su código JavaScript.

Ahora que ya sabes cómo utilizar los componentes web, las siguientes tres secciones cubren las tres características del navegador web que nos permiten implementarlos.

#### NODOS DE FRAGMENTACIÓN DE DOCUMENTOS

Antes de que podamos cubrir las APIs de componentes web, necesitamos volver brevemente a la API DOM para explicar qué es un DocumentFragment. La API del DOM organiza un documento en un árbol de objetos Node, donde un Node puede ser un Documento, un Elemento, un nodo de Texto, o incluso un Comentario. Ninguno de estos tipos de nodos permite representar un fragmento de un documento que consiste en un conjunto de nodos hermanos sin su padre. Aquí es donde entra DocumentFragment: es otro tipo de Nodo que sirve como padre temporal cuando se quiere manipular un grupo de nodos hermanos como una sola unidad. Puedes crear un nodo DocumentFragment con `document.createDocumentFragment()`. Una vez que tenga un DocumentFragment, puede utilizarlo como un elemento y añadirle contenido. Un DocumentFragment es diferente de un Elemento porque no tiene un padre. Pero lo más importante es que cuando se inserta un nodo DocumentFragment en el documento, el DocumentFragment mismo no se inserta. En cambio, se insertan todos sus hijos.

#### 15.6.2 Plantillas HTML

La etiqueta HTML `<template>` sólo está relacionada con los componentes web, pero permite una optimización útil para los componentes que aparecen con frecuencia en las páginas web. Las etiquetas `<template>` y sus hijos nunca son representados por un navegador web y sólo son útiles en las páginas web que utilizan JavaScript. La idea detrás de esta etiqueta es que cuando una página web contiene múltiples repeticiones de la misma estructura HTML básica (como las filas de una tabla o la implementación interna de un componente web), entonces podemos usar una `<plantilla>` para definir la estructura de ese elemento una vez, y luego usar JavaScript para duplicar la estructura tantas veces como sea necesario.

En JavaScript, una etiqueta `<plantilla>` está representada por un

Objeto `HTMLTemplateElement`. Este objeto define una única propiedad de contenido, y el valor de esta propiedad es un `DocumentFragment` de todos los nodos hijos de la `<plantilla>`. Puede clonar este `DocumentFragment` y luego insertar la copia clonada en su documento según sea necesario. El fragmento en sí no se insertará, pero sí sus hijos. Suponga que está trabajando con un documento que incluye una etiqueta `<table>` y `<template id="row">` y que la plantilla define la estructura de las filas de esa tabla. Podrías usar la plantilla así:

```
let tableBody = document.querySelector("tbody"); let template = document.querySelector("#row"); let clone = template.content.cloneNode(true); // clon profundo  
// ...Usa el DOM para insertar contenido en los elementos <td> del clon... // Ahora añade la fila clonada e inicializada en la tabla tableBody.append(clone);
```

Los elementos de plantilla no tienen que aparecer literalmente en un documento HTML para ser útiles. Puedes crear una plantilla en tu código JavaScript, crear sus hijos con `innerHTML`, y luego hacer tantos clones como sea necesario sin la sobrecarga de análisis de `innerHTML`. Esta es la forma en que las plantillas HTML se utilizan típicamente en los componentes web, y [el Ejemplo 15-3](#) demuestra esta técnica.

### 15.6.3 Elementos personalizados

La segunda característica del navegador web que permite los componentes web es

"elementos personalizados": la capacidad de asociar una clase de JavaScript con un nombre de etiqueta HTML para que cualquier etiqueta de este tipo en el documento se convierta automáticamente en instancias de la clase en el árbol DOM. El método `customElements.define()` toma un nombre de etiqueta de componente web como primer argumento (recuerde que el nombre

de la etiqueta debe incluir un guión) y una subclase de HTMLElement como segundo argumento. Cualquier elemento existente en el documento con ese nombre de etiqueta es "actualizado" a instancias recién creadas de la clase. Y si el navegador analiza cualquier HTML en el futuro, creará automáticamente una instancia de la clase para cada una de las etiquetas que encuentre.

La clase pasada a customElements.define() debe extender HTMLElement y no un tipo más específico como HTMLButtonElement.<sup>4</sup> Recordemos del [capítulo 9](#) que cuando una clase JavaScript extiende otra clase, la función constructora debe llamar a super() antes de utilizar la palabra clave this, por lo que si la clase de elemento personalizado tiene un constructor, debe llamar a super() (sin argumentos) antes de hacer cualquier otra cosa.

El navegador invocará automáticamente ciertos "métodos del ciclo de vida" de una clase de elemento personalizado. El método connectedCallback() se invoca cuando una instancia del elemento personalizado se inserta en el documento, y muchos elementos utilizan este método para realizar la inicialización. También hay un método disconnectedCallback() que se invoca cuando (y si) el elemento se retira del documento, aunque se utiliza con menos frecuencia.

Si una clase de elemento personalizado define una propiedad estática observedAttributes cuyo valor es una matriz de nombres de atributos, y si alguno de los atributos nombrados se establece (o cambia) en una instancia del elemento personalizado, el navegador invocará la función

`attributeChangedCallback()`, pasando el atributo nombre, su valor anterior y su nuevo valor. Esta llamada de retorno puede tomar los pasos necesarios para actualizar el componente basándose en los valores de sus atributos.

Las clases de elementos personalizados también pueden definir cualquier otra propiedad y método que deseen. Por lo general, definirán métodos getter y setter que hagan que los atributos del elemento estén disponibles como propiedades de JavaScript.

Como ejemplo de elemento personalizado, supongamos que queremos ser capaces de mostrar círculos dentro de párrafos de texto normal. Nos gustaría poder escribir HTML como éste para representar problemas de historias matemáticas como el que se muestra en la [Figura 15-4](#):

`<p>`

El documento tiene una canica: `<inline-circle></inlinecircle>`.

El analizador HTML instala dos canicas más:

`<inline-circle diameter="1.2em" color="blue"></inlinecircle> <inline-circle diameter=".6em" color="gold"></inlinecircle>`.

¿Cuántas canicas contiene ahora el documento?

`</p>`

The document has one marble: . The HTML parser instantiates two more marbles:  . How many marbles does the document contain now?

*Figura 15-4. Un elemento personalizado de círculo en línea*

Podemos implementar este elemento personalizado `<inline-circle>` con el código mostrado en el [Ejemplo 15-2](#):

*Ejemplo 15-2. El elemento personalizado `<inline-circle>`.*

---

```
customElements.define("inline-circle", class InlineCircle extends HTMLElement {  
    // El navegador llama a este método cuando un elemento <inline-circle>.  
    // se inserta en el documento. También hay un disconnectedCallback() //  
    // que no necesitamos en este ejemplo. connectedCallback() { // Establecer los  
    // estilos necesarios para crear círculos this.style.display = "inline-block"; this.  
    // style.borderRadius = "50%"; this.style.border = "solid black 1px"; this.  
    // style.transform = "translateY(10%)";
```

```
// Si aún no hay un tamaño definido, establece un tamaño por defecto //  
que se basa en el tamaño de la fuente actual. if (! this. style. width) { this. style.  
width = "0.8em"; this. style. height = "0.8em"; }  
}  
  
// La propiedad estática observedAttributes especifica qué atributos  
// queremos que se nos notifiquen los cambios. (Usamos un getter aquí ya que  
// sólo podemos usar "static" con métodos). static get observedAttributes() {  
return ["diameter", "color"]; }  
  
// Esta llamada de retorno es invocada cuando uno de los atributos listados arriba //  
cambia, ya sea cuando el elemento personalizado es analizado por primera vez, o después.  
attributeChangedCallback(name, oldValue, newValue) { switch(name) { case  
"diameter":  
    // Si el atributo de diámetro cambia, actualiza los estilos de tamaño this. style.  
    width = newValue; this. style. height = newValue; break; case "color":  
    // Si el atributo de color cambia, actualiza los estilos de color this. style.  
    backgroundColor = newValue; break; }  
}  
  
// Definir las propiedades de JavaScript que corresponden al elemento  
// atributos. Estos getters y setters sólo obtienen y establecen los atributos subyacentes  
// atributos. Si se establece una propiedad de JavaScript, que establece el atributo  
// que desencadena una llamada a attributeChangedCallback()
```

que actualiza // los estilos de los elementos.

```
get diameter() { return this.getAttribute("diameter"); } set diameter(diameter) {  
this.setAttribute("diameter", diameter); } get color() { return this.  
getAttribute("color"); } set color(color) { this.setAttribute("color", color); }  
});
```

#### 15.6.4 DOM en la sombra

El elemento personalizado demostrado en el [Ejemplo 15-2](#) no está bien encapsulado. Cuando estableces sus atributos de diámetro o color, responde alterando su propio atributo de estilo, lo cual no es un comportamiento que esperaríamos de un elemento HTML real. Para convertir un elemento personalizado en un verdadero componente web, debería utilizar el poderoso mecanismo de encapsulación conocido como *shadow DOM*.

Shadow DOM permite adjuntar una "raíz sombra" a un elemento personalizado (y también a un elemento <div>, <span>, <body>, <article>, <main>, <nav>, <header>, <footer>, <section>, <p>, <blockquote>, <aside>, o <h1> hasta <h6> elemento) conocido como "shadow host". Los elementos host sombra, como todos los elementos HTML, ya son la raíz de un árbol DOM normal de elementos descendientes y nodos de texto. Una raíz en la sombra es la raíz de otro árbol más privado de elementos descendientes que brota del anfitrión en la sombra y puede considerarse como un minidocumento distinto.

La palabra "sombra" en "DOM sombra" se refiere al hecho de que los elementos que descienden de una raíz sombra se "esconden en las sombras": no forman parte del árbol DOM normal, no aparecen en la matriz de hijos de su elemento anfitrión y no son visitados por los

métodos normales de recorrido del DOM, como `querySelector()`. Por el contrario, los hijos normales y regulares del DOM de un anfitrión en la sombra se denominan a veces el "DOM ligero".

Para entender el propósito del DOM en la sombra, imagina los elementos HTML `<audio>` y `<video>`: muestran una interfaz de usuario no trivial para controlar la reproducción de medios, pero los botones de reproducción y pausa y otros elementos de la IU no forman parte del árbol del DOM y no pueden ser manipulados por JavaScript. Dado que los navegadores web están diseñados para mostrar HTML, es natural que los proveedores de navegadores quieran mostrar interfaces de usuario internas como éstas utilizando HTML. De hecho, la mayoría de los navegadores llevan mucho tiempo haciendo algo así, y el shadow DOM lo convierte en una parte estándar de la plataforma web.

## ENCAPSULACIÓN SHADOW DOM

La característica clave del shadow DOM es la encapsulación que proporciona. Los descendientes de una raíz sombra están ocultos -e independientes- del árbol DOM normal, casi como si estuvieran en un documento independiente. Hay tres tipos de encapsulación muy importantes que proporciona el shadow DOM:

- Como ya se ha mencionado, los elementos del DOM sombra están ocultos a los métodos regulares del DOM como `querySelectorAll()`. Cuando se crea una raíz sombra y se adjunta a su host sombra, puede crearse en modo "abierto" o "cerrado". Una raíz sombra cerrada está completamente sellada e inaccesible. Sin embargo, lo más común es que las raíces sombra se creen en modo "abierto", lo que significa que el host sombra tiene una propiedad `shadowRoot` que JavaScript puede utilizar para

acceder a los elementos de la raíz sombra, si tiene alguna razón para hacerlo.

- Los estilos definidos bajo una raíz de sombra son privados para ese árbol y nunca afectarán a los elementos DOM ligeros del exterior. (Una raíz de sombra puede definir estilos por defecto para su elemento anfitrión, pero éstos serán anulados por los estilos DOM ligeros). Del mismo modo, los estilos DOM ligeros que se aplican al elemento anfitrión de la sombra no tienen efecto en los descendientes de la raíz de la sombra. Los elementos en el DOM de sombra heredarán cosas como el tamaño de la fuente y el color de fondo del DOM de luz, y los estilos en el DOM de sombra pueden optar por utilizar variables CSS definidas en el DOM de luz. Sin embargo, en su mayor parte, los estilos del DOM de la luz y los estilos del DOM de la sombra son completamente independientes: el autor de un componente web y el usuario de un componente web no tienen que preocuparse por colisiones o conflictos entre sus hojas de estilo. La posibilidad de "abrir" el CSS de esta manera es quizás la característica más importante del shadow DOM.
- Algunos eventos (como "load") que ocurren dentro del DOM en la sombra se limitan al DOM en la sombra. Otros, como los eventos de enfoque, ratón y teclado, se propagan hacia arriba y hacia afuera. Cuando un evento que se origina en el DOM de la sombra cruza el límite y comienza a propagarse en el DOM de la luz, su propiedad de destino se cambia al elemento anfitrión de la sombra, por lo que parece haberse originado directamente en ese elemento.

## RANURAS DE DOMO DE SOMBRA Y NIÑOS DE DOMO DE LUZ

Un elemento HTML que es un host de sombra tiene dos árboles de descendientes. Uno es la matriz `children[]` -los descendientes regulares del elemento anfitrión- y el otro es la raíz de la sombra y todos sus descendientes, y puede que te preguntes cómo se pueden mostrar dos árboles de contenido distintos dentro del mismo elemento anfitrión. Así es como funciona:

- Los descendientes de la raíz sombra siempre se muestran dentro del host sombra.
- Si esos descendientes incluyen un elemento <slot>, entonces los hijos light DOM regulares del elemento anfitrión se muestran como si fueran hijos de ese <slot>, reemplazando cualquier contenido shadow DOM en el slot. Si el shadow DOM no incluye un <slot>, cualquier contenido light DOM del host no se muestra nunca. Si el shadow DOM tiene un <slot>, pero el shadow host no tiene hijos light DOM, entonces el contenido shadow DOM del slot se muestra por defecto.
- Cuando el contenido del DOM ligero se muestra dentro de un espacio del DOM sombra, decimos que esos elementos han sido "distribuidos", pero es importante entender que los elementos no se convierten realmente en parte del DOM sombra. Todavía pueden ser consultados con `querySelector()`, y siguen apareciendo en el light DOM como hijos o descendientes del elemento anfitrión.
- Si el shadow DOM define más de una <slot> y nombra esas ranuras con un atributo `name`, los hijos del shadow host pueden especificar en qué ranura quieren aparecer especificando un atributo `slot="slotname"`. Vimos un ejemplo de este uso en [§15.6.1](#) cuando demostramos cómo personalizar los iconos mostrados por el componente <búsqueda>.

## API SHADOW DOM

A pesar de toda su potencia, el Shadow DOM no tiene mucho API de JavaScript. Para convertir un elemento DOM ligero en un host de sombra, basta con llamar a su método `attachShadow()`, pasando `{mode: "open"}` como único argumento. Este método devuelve un objeto raíz de sombra y también establece ese objeto como el valor de la propiedad `shadowRoot` del anfitrión. El objeto shadow root es

un DocumentFragment, y puedes usar métodos DOM para añadirle contenido o simplemente establecer su propiedad innerHTML a una cadena de HTML.

Si tu componente web necesita saber cuándo ha cambiado el contenido del DOM de luz de una <ranura> del DOM de sombra, puede registrar un oyente para los eventos "slotchanged" directamente en el elemento <ranura>.

### **15.6.5 Ejemplo: un componente web < search-box>**

La figura 15-3 ilustra un componente web < search-box>.

El ejemplo 15-3 demuestra las tres tecnologías de habilitación que definen los componentes web: implementa el componente < search-box> como un elemento personalizado que utiliza una etiqueta < template> para la eficiencia y una raíz de sombra para la encapsulación.

Este ejemplo muestra cómo utilizar directamente las API de bajo nivel de los componentes web. En la práctica, muchos componentes web desarrollados hoy en día los crean utilizando bibliotecas de alto nivel como "lit-element". Una de las razones para usar una librería es que crear componentes reutilizables y personalizables es en realidad bastante difícil de hacer bien, y hay muchos detalles que hacer bien. El ejemplo 15-3 muestra componentes web y hace un manejo básico del enfoque del teclado, pero por lo demás ignora la accesibilidad y no hace ningún intento de utilizar los atributos ARIA adecuados para que el componente funcione con lectores de pantalla y otras tecnologías de asistencia.

*Ejemplo 15-3. Implementación de un componente web*

```
/*
 * Esta clase define un elemento HTML <search-box> personalizado que muestra
un
 * <input> campo de entrada de texto más dos iconos o emoji. Por defecto,
muestra un
 * emoji de lupa (que indica búsqueda) a la izquierda del campo de texto
 * y un emoji X (que indica cancelación) a la derecha del campo de texto. En
 * oculta el borde del campo de entrada y muestra un borde alrededor de sí mismo,
 * creando la apariencia de que los dos emoji están dentro de la entrada
 * campo. Del mismo modo, cuando se enfoca el campo de entrada interno, el
anillo de enfoque
 * se muestra alrededor del <buzón de búsqueda>.
*
 * Puede anular los iconos por defecto incluyendo <span> o
<img> niños
 * de <search-box> con los atributos slot="left" y slot="right".
*
 * <búsqueda> admite los atributos normales de HTML desactivado y oculto y
 * también los atributos size y placeholder, que tienen el mismo significado para
este
 * como lo hacen para el elemento <input>.
*
 * Los eventos de entrada del elemento interno <input> burbujean y aparecen con
 * su campo de destino establecido en el elemento <search-box>.
*
 * El elemento dispara un evento de "búsqueda" con la propiedad detail
establecida en el
 * cadena de entrada actual cuando el usuario hace clic en el emoji de la izquierda
(la lupa
 * de vidrio). El evento "búsqueda" también se envía cuando el campo de texto
interno
 * genera un evento de "cambio" (cuando el texto ha cambiado y el usuario escribe
 * Retorno o Tab).
*
 * El elemento dispara un evento "clear" cuando el usuario hace clic en el emoji
correcto
 * (la X). Si ningún controlador llama a preventDefault() en el evento, el elemento
 * borra la entrada del usuario una vez que se ha completado el envío del evento.
*
 * Tenga en cuenta que no hay propiedades o atributos onsearch y onclear:
```

```
*      los manejadores de los eventos "search" y "clear" sólo pueden registrarse con
*      addEventListener().
*/
class SearchBox extends HTMLElement { constructor() { super(); // Invoca el
constructor de la superclase; debe ser el primero.

    // Crea un árbol DOM de sombra y lo adjunta a este elemento, estableciendo
    // el valor de this.shadowRoot. this. attachShadow({mode: "open"});

    // Clona la plantilla que define los descendientes y la hoja de estilos para // este
componente personalizado, y añade ese contenido a la raíz de la sombra.
this. shadowRoot. append(SearchBox. template. content. cloneNode(true ));

    // Obtener referencias a los elementos importantes en el shadow DOM this. input =
this. shadowRoot. querySelector("#input"); let leftSlot = this. shadowRoot.
querySelector('slot[name="left"]'); let rightSlot = this. shadowRoot.
querySelector('slot[name="right"]');

    // Cuando el campo de entrada interno obtiene o pierde el foco, establece o elimina
    // el atributo "focused" que hará que nuestra hoja de estilo interna
    // para mostrar u ocultar un anillo de enfoque falso en todo el componente. Nota
    // que los eventos "blur" y "focus" burbujeen y parezcan originarse // de la <search-
box>.
this. input. onfocus = () => { this. setAttribute("focused", ""); }; this.
input. onblur = () => { this. removeAttribute("focused");};

    // Si el usuario hace clic en la lupa, dispara
```

```
una "búsqueda"
    // evento. También se dispara si el campo de entrada dispara un
    "cambio"
        // evento. (El evento "change" no sale del DOM de la sombra). leftSlot. onclick = this.
input. onchange = (event) => { event.stopPropagation(); // Evitar que los eventos de click
salgan si (this. disabled) return; // No hacer nada cuando se desactiva this.
dispatchEvent(new CustomEvent("search", { detail: this. input. value }));
};

// Si el usuario hace clic en la X, dispara un evento "clear".      // Si no se llama
a preventDefault() en el evento, borra la entrada.
rightSlot. onclick = (event) => { event.stopPropagation(); // No dejar que el clic se eleve
si (this. disabled) return; // No hacer nada si se desactiva
    let e = new CustomEvent("clear", { cancelable: true });
    this. dispatchEvent(e); if (! e. defaultPrevented) { // Si el evento no fue "cancelado"
this. input. value = ""; // entonces borra el campo de entrada }
};

// Cuando algunos de nuestros atributos se establecen o cambian, necesitamos establecer
el
// valor correspondiente en el elemento interno <input>.
Este ciclo de vida
// junto con la propiedad estática observedAttributes de abajo, // se encarga de
ello.
attributeChangedCallback(name, oldValue, newValue) {
```

```
if (name === "disabled") { this.input.disabled = newValue !== null; }
else if (name === "placeholder") { this.input.placeholder = newValue; }
else if (name === "size") { this.input.size = newValue; } else if (name ===
"value") { this.input.value = newValue; }

// Por último, definimos los getters y setters de las propiedades que
// corresponden a los atributos HTML que soportamos. Los getters simplemente
devuelven
// el valor (o la presencia) del atributo. Y los setters sólo establecen
// el valor (o la presencia) del atributo. Cuando un método setter
// cambia un atributo, el navegador invocará automáticamente el // attributeChangedCallback anterior.

get placeholder() { return this.getAttribute("placeholder"); } get size() { return this.
getAttribute("size"); } get value() { return this.getAttribute("value"); } get disabled() {
return this.hasAttribute("disabled"); } get hidden() { return this.
hasAttribute("hidden"); }

set placeholder(value) { this.setAttribute("placeholder", value); } set size(value) {
this.setAttribute("size", value); } set value(text) { this.setAttribute("value", text); } set
disabled(value) { if (value) this.setAttribute("disabled", ""); else this.
removeAttribute("disabled"); } set hidden(value) { if (value) this.setAttribute("hidden",
 ""); else this.removeAttribute("hidden"); }
```

```
    }
}

// Este campo estático es necesario para el método attributeChangedCallback. // Sólo
los atributos nombrados en este array activarán las llamadas a ese método.
SearchBox. observedAttributes = ["disabled", "placeholder",
"tamaño", "valor"];

// Crear un elemento <template> para contener la hoja de estilos y el árbol de //
elementos que usaremos para cada instancia del elemento SearchBox. SearchBox.
template = document.createElement("template");

// Inicializamos la plantilla analizando esta cadena de HTML.
Sin embargo, tenga en cuenta,
// que cuando instanciamos un SearchBox, podemos simplemente clonar los nodos //
en la plantilla y no tenemos que parsear el HTML de nuevo.
SearchBox. template. innerHTML = `

<estilo>
/*
*      El selector :host hace referencia al elemento <search-box> de la luz
*      DOM. Estos estilos son predeterminados y pueden ser anulados por el usuario del
*      <búsqueda-box> con estilos en el DOM ligero.
*/
:host {
    display: inline-block; /* El valor por defecto es inline */ border: solid black 1px; /* Un
    borde redondeado alrededor del
<input> y <slots> */ border-radius: 5px; padding: 4px 6px; /* Y algo de espacio dentro del
borde
*/
}
:host([hidden]) { /* Observe el paréntesis: cuando el host tiene hidden... */ display:none;
/* ... atributo establecido no se muestra
*/
}
:host([disabled]) { /* Cuando el host tiene el atributo disabled... */ opacidad:
0.5; /* ...grisáceo */}
```

```

}

:host([focused]) { /* Cuando el host tiene el atributo focused... */ box-shadow: 0 0
2px 2px #6AE; /* mostrar este anillo de enfoque falso. */ }

/* El resto de la hoja de estilos sólo se aplica a los elementos del
Shadow DOM. */ input { border-width: 0; /* Ocultar el borde del campo de entrada
interno. */ outline: none; /* Ocultar también el anillo de enfoque. */ font: inherit; /*
Los elementos <input> no heredan la fuente por defecto */ background: inherit; /* Lo
mismo para el color de fondo. */

} slot { cursor: default; /* Un cursor de flecha sobre los botones */ user-select: none; /* No
dejar que el usuario seleccione el texto del emoji */
}

</style>
<div>
  <slot name="left"> \u{1f50d}</slot> <!-- U+1F50D es una lupa -->
  <input type="text" id="input" /> <!-- El elemento de entrada real -->
  <slot name="right"> \u{2573}</slot> <!-- U+2573 es una X -->
</div> <div>
`;

// Por último, llamamos a customElement.define() para registrar el
Elemento SearchBox
// como la implementación de la etiqueta <search-box>. Los elementos
personalizados deben // tener un nombre de etiqueta que contenga un guión.
customElements.define("search-box", SearchBox);

```

## 15.7 SVG: gráficos vectoriales escalables

SVG (scalable vector graphics) es un formato de imagen. La palabra "vector" en su nombre indica que es fundamentalmente diferente de los formatos de imagen de trama, como GIF, JPEG y PNG, que especifican una matriz de valores de píxeles. En su lugar, una "imagen" SVG es una descripción precisa e independiente de la

resolución (por tanto, "escalable") de los pasos necesarios para dibujar el gráfico deseado. Las imágenes SVG se describen mediante archivos de texto que utilizan el lenguaje de marcado XML, que es bastante similar al HTML.

Hay tres maneras de utilizar SVG en los navegadores web:

1. Puede utilizar archivos de imagen `.svg` con las etiquetas HTML `<img>` normales, igual que utilizaría una imagen `.png` o `.jpeg`.
2. Dado que el formato SVG basado en XML es tan similar al HTML, puede incrustar etiquetas SVG directamente en sus documentos HTML. Si lo hace, el analizador HTML del navegador le permite omitir los espacios de nombres XML y tratar las etiquetas SVG como si fueran etiquetas HTML.
3. Puede utilizar la API del DOM para crear dinámicamente elementos SVG que generen imágenes bajo demanda.

Las subsecciones siguientes muestran el segundo y tercer uso de SVG. Tenga en cuenta, sin embargo, que SVG tiene una gramática amplia y moderadamente compleja. Además de las primitivas de dibujo de formas simples, incluye soporte para curvas arbitrarias, texto y animación. Los gráficos SVG pueden incluso incorporar scripts de JavaScript y hojas de estilo CSS para añadir información de comportamiento y presentación. Una descripción completa de SVG va mucho más allá del alcance de este libro. El objetivo de esta sección es sólo mostrarle cómo puede utilizar SVG en sus documentos HTML y programarlo con JavaScript.

### **15.7.1 SVG en HTML**

Las imágenes SVG pueden, por supuesto, mostrarse utilizando las etiquetas HTML < img>. Pero también puede incrustar SVG directamente en HTML. Y si lo hace, puede incluso utilizar hojas de estilo CSS para especificar cosas como fuentes, colores y anchos de línea. Aquí, por ejemplo, hay un archivo HTML que utiliza SVG para mostrar la esfera de un reloj analógico:

```
< html>
<cabeza>
< title> Reloj analógico</title>
<estilo> /* Estos estilos CSS se aplican todos a los elementos SVG definidos a continuación
 * / #reloj { /* Estilos para todo lo que hay en el reloj: */ trazo: negro; /* líneas negras */
trazo-tapa: redondo; /* con extremos redondeados */ relleno: #ffe; /* sobre fondo
blanco */ }
#clock .face { stroke-width: 3; } /* Contorno de la cara del reloj */
#clock .ticks { stroke-width: 2; } /* Líneas que marcan cada hora */ #clock .hands {
stroke-width: 3; } /* Cómo dibujar las manecillas del reloj */ #clock .numbers { /* Cómo
dibujar los números */ font-family: sans-serif; font-size: 10; font-weight:
negrita; text-anchor: middle; stroke: none; fill: black;
}
</style>
</head>
<body> < svg id="reloj" viewBox="0 0 100 100" width="250"
height="250">
    <!-- Los atributos de anchura y altura son el tamaño de la pantalla del gráfico --&gt;
    <!-- El atributo viewBox da el sistema de coordenadas interno --&gt;
    &lt;círculo class="face" cx="50" cy="50" r="45"/&gt; &lt;!-- la esfera del reloj --&gt;</pre>
```

```

<g class="ticks"> <!-- marcas de verificación para cada una de las 12 horas -->
  <línea x1='50' y1='5.000' x2='50.00' y2='10.00' />
  <línea x1='72.50' y1='11.03' x2='70.00' y2='15.36' />
  <línea x1='88.97' y1='27.50' x2='84.64' y2='30.00' />
  <línea x1='95.00' y1='50.00' x2='90.00' y2='50.00' />
  <línea x1='88.97' y1='72.50' x2='84.64' y2='70.00' />
  <línea x1='72.50' y1='88.97' x2='70.00' y2='84.64' />
  <línea x1='50,00' y1='95,00' x2='50,00' y2='90,00' />
  <línea x1='27.50' y1='88.97' x2='30.00' y2='84.64' />
  <línea x1='11.03' y1='72.50' x2='15.36' y2='70.00' />
  <línea x1='5.000' y1='50.00' x2='10.00' y2='50.00' />
  <línea x1='11.03' y1='27.50' x2='15.36' y2='30.00' /> <línea x1='27.50' y1='11.03'
    x2='30.00' y2='15.36' />
</g> <g class="numbers"> <!-- Numera los puntos cardinales-->
< texto x="50" y="18"> 12</texto>< texto x="85" y="53"> 3</texto>
< texto x="50" y="88"> 6</texto>< texto x="15" y="53"> 9</texto> </g> <g
class="hands"> <!-- Dibuja las manos apuntando hacia arriba.
-->
  <línea class="hourhand" x1="50" y1="50" x2="50" y2="25" />
  <línea class="minutehand" x1="50" y1="50" x2="50" y2="20" /> <g>
</svg>
<script src="clock.js"></script>
</body>
</html>

```

Observará que los descendientes de la etiqueta `< svg >` no son etiquetas HTML normales. Las etiquetas `< circle >`, `< line >` y `< text >` tienen propósitos obvios, sin embargo, y debería estar claro cómo funciona este gráfico SVG. Sin embargo, hay muchas otras etiquetas SVG, y tendrá que consultar una referencia de SVG para saber más. También puede notar que la hoja de estilos es extraña. Los estilos como `fill`, `stroke-width` y `text-anchor` no son propiedades de estilo CSS normales. En este caso, CSS se utiliza esencialmente para establecer atributos de las etiquetas SVG que aparecen en el documento. Tenga en cuenta también que la propiedad abreviada de fuente de CSS no funciona para las etiquetas SVG, y debe

establecer explícitamente font-family, font-size y font-weight como propiedades de estilo independientes.

### 15.7.2 Scripting SVG

Una de las razones para incrustar SVG directamente en sus archivos HTML (en lugar de utilizar etiquetas `<img>` estáticas) es que si lo hace, puede utilizar la API DOM para manipular la imagen SVG.

Supongamos que utiliza SVG para mostrar iconos en su aplicación web. Podría incrustar el SVG dentro de una etiqueta `<template>` ([§15.6.2](#)) y luego clonar el contenido de la plantilla cada vez que necesite insertar una copia de ese ícono en su interfaz de usuario. Y si quieras que el ícono responda a la actividad del usuario - cambiando de color cuando el usuario pasa el puntero por encima, por ejemplo-, a menudo puedes conseguirlo con CSS.

También es posible manipular dinámicamente los gráficos SVG que están directamente incrustados en HTML. El ejemplo de la esfera del reloj de la sección anterior muestra un reloj estático con las manecillas de las horas y los minutos orientadas hacia arriba que muestran la hora del mediodía o la medianoche. Pero habrá notado que el archivo HTML incluye una etiqueta `<script>`. Ese script ejecuta una función periódicamente para comprobar la hora y transformar las manecillas de las horas y los minutos girándolas el número apropiado de grados para que el reloj muestre realmente la hora actual, como se muestra en la [Figura 155](#).

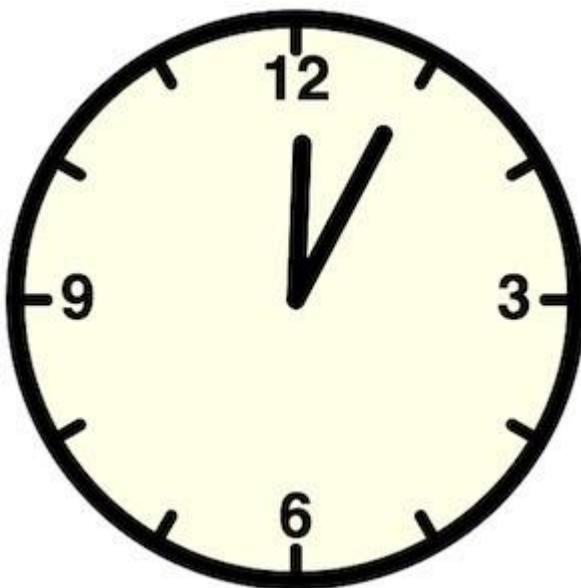


Figura 15-5. Un reloj analógico SVG con guión

El código para manipular el reloj es sencillo. Determina el ángulo adecuado de las manecillas de las horas y los minutos basándose en la hora actual, luego utiliza querySelector() para buscar los elementos SVG que muestran esas manecillas, y luego establece un atributo de transformación en ellos para girarlos alrededor del centro de la esfera del reloj. La función utiliza setTimeout() para asegurar que se ejecuta una vez por minuto:

```
(function updateClock() { // Actualizar el gráfico del reloj SVG para mostrar la hora actual
let now = new Date(); // Hora actual let sec = now.getSeconds(); // Segundos let min =
now.getMinutes() + sec/60; // Minutos fraccionados let hour = (now.getHours() % 12) +
min/60; // Horas fraccionadas let minangle = min * 6; // 6 grados por minuto let hourangle
= hour * 30; // 30 grados por hora
```

```
// Obtener elementos SVG para las manecillas del reloj let minhand =
document.querySelector("#clock . minutehand");
let hourhand = document.querySelector("#reloj . hourhand");

// Establecer un atributo SVG en ellos para moverlos alrededor de la esfera del reloj
minhand.setAttribute("transform", `rotate(${minangle},50,50)`); hourhand.
setAttribute("transform", `rotate(${hourangle},50,50)`);

// Ejecute esta función de nuevo en 10 segundos setTimeout(updateClock,
10000); }()); // Observe la invocación inmediata de la función aquí.
```

### 15.7.3 Creación de imágenes SVG con JavaScript

Además de la simple programación de imágenes SVG incrustadas en sus documentos HTML, también puede crear imágenes SVG desde cero, lo que puede ser útil para crear visualizaciones de datos cargados dinámicamente, por ejemplo. [El Ejemplo 15-4](#) muestra cómo puede utilizar JavaScript para crear gráficos circulares SVG, como el que se muestra en la [Figura 15-6](#).

Aunque las etiquetas SVG pueden incluirse en documentos HTML, técnicamente son etiquetas XML, no HTML, y si desea crear elementos SVG con la API DOM de JavaScript, no puede utilizar la función normal createElement() que se introdujo en [§15.3.5](#). En su lugar, debe utilizar createElementNS(), que toma una etiqueta XML como primer argumento. En el caso de SVG, ese espacio de nombres es la cadena literal "http://www.w3.org/2000/svg".

## Programming languages by percentage of professional developers who report their use

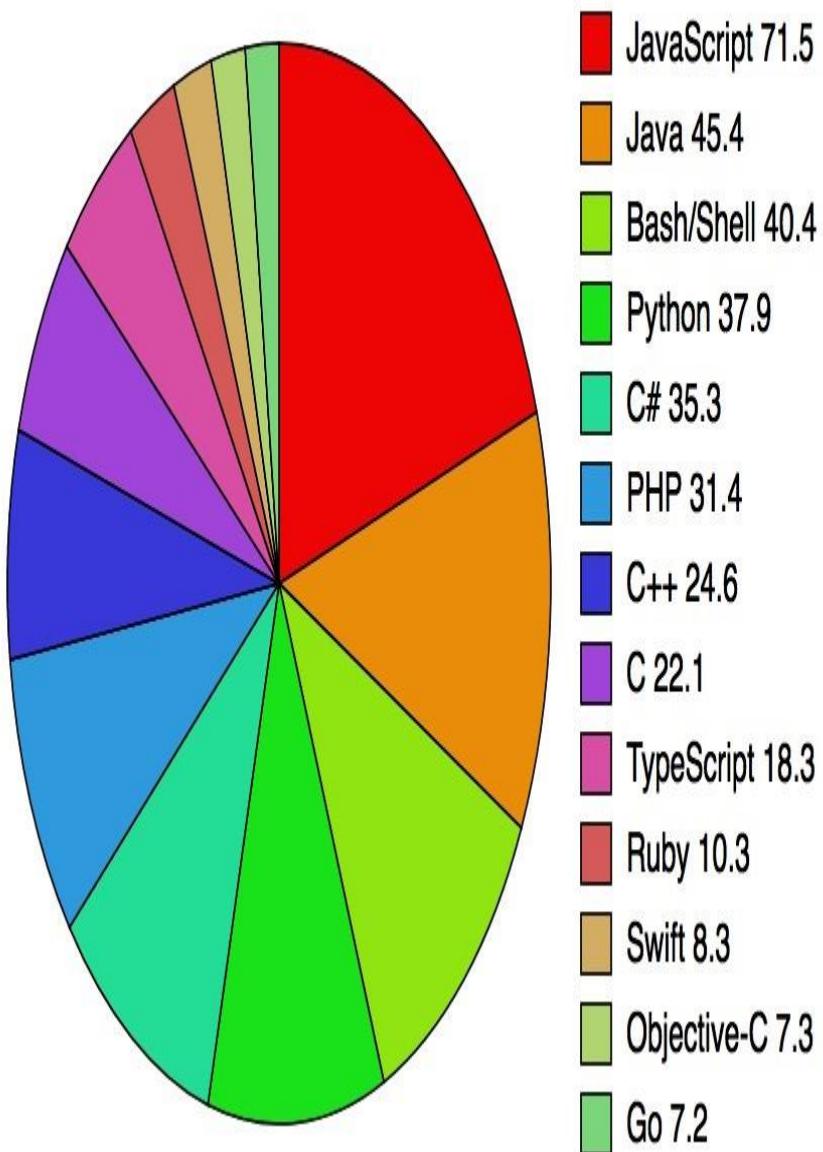


Figura 15-6. Un gráfico circular SVG construido con JavaScript (datos de la encuesta de desarrolladores de 2018 de Stack Overflow sobre las tecnologías más populares)

A parte del uso de `createElementNS()`, el código para dibujar el gráfico circular del [Ejemplo 15-4](#) es relativamente sencillo. Hay un poco de matemática para convertir los datos que se grafican en ángulos de cortes de tarta. La mayor parte del ejemplo, sin

embargo, es código DOM que crea elementos SVG y establece atributos en esos elementos.

La parte más opaca de este ejemplo es el código que dibuja las rebanadas reales del pastel. El elemento que se utiliza para mostrar cada rebanada es < path>. Este

El elemento SVG describe formas arbitrarias compuestas por líneas y curvas. La descripción de la forma se especifica mediante el atributo d del elemento < path>. El valor de este atributo utiliza una gramática compacta de códigos de letras y números que especifican coordenadas, ángulos y otros valores. La letra M, por ejemplo, significa "mover a" y va seguida de las coordenadas x e y. La letra L significa "línea a" y dibuja una línea desde el punto actual hasta las coordenadas que le siguen. Este ejemplo también utiliza la letra A para dibujar un arco. Esta letra va seguida de siete números que describen el arco, y puedes buscar la sintaxis en Internet si quieres saber más.

*Ejemplo 15-4. Dibujar un gráfico circular con JavaScript y SVG*

---

```
/** * Crear un elemento <svg> y dibujar un gráfico circular en él.  
*  
*      Esta función espera un argumento de objeto con las siguientes propiedades:  
*  
*      anchura, altura: el tamaño del gráfico SVG, en píxeles  
*      cx, cy, r: el centro y el radio de la tarta  
*      lx, ly: la esquina superior izquierda de la leyenda del gráfico  
*      datos: un objeto cuyos nombres de propiedades son etiquetas de datos y cuyo  
*              los valores de las propiedades son los valores asociados a cada etiqueta  
*  
*      La función devuelve un elemento <svg>. La persona que lo llama debe insertarlo  
en  
*      el documento para hacerlo visible.  
*/function pieChart(options) {
```



Suscríbete a DeepL Pro para poder editar este documento.  
Entra en [www.DeepL.com/pro](https://www.DeepL.com/pro) para más información.

```

let {width, height, cx, cy, r, lx, ly, data} = options;

// Este es el espacio de nombres XML para los elementos svg let svg =
"http://www.w3.org/2000/svg";

// Crear el elemento <svg>, y especificar el tamaño de los píxeles y las coordenadas
// del usuario let chart = document.createElementNS(svg, "svg"); chart.
setAttribute("width", width); chart.setAttribute("height", height); chart.
setAttribute("viewBox", `0 0 ${width} ${height}`);

// Define los estilos de texto que usaremos para el gráfico. Si dejamos estos // valores
// sin establecer aquí, pueden ser establecidos con CSS en su lugar.
chart.setAttribute("font-family", "sans-serif"); chart.setAttribute("font-size", "18");

// Obtenga las etiquetas y los valores como arrays y sume los valores para saber cómo // es de grande el pastel.
let labels = Object.keys(data); let values = Object.
values(data); let total = values.reduce((x,y) => x+y);

// Calcula los ángulos de todas las rodajas. La rebanada i comienza en ángulos[i] // y
// termina en ángulos[i+1]. Los ángulos se miden en radianes.
let angles = [0]; values.forEach((x, i) => angles.push(angles[i] + x/total * 2 * Math.PI));

// Ahora recorre en bucle los valores de la tarta. forEach((valor, i) => {
  // Calcula los dos puntos en los que nuestra rebanada interseca el círculo
  // Estas fórmulas están elegidas de manera que un ángulo de 0 está a las 12 horas
  // y los ángulos positivos aumentan en el sentido de las agujas del reloj.
  let x1 = cx + r * Math.sin(ángulos[i]); let y1 = cy - r * Math.
cos(ángulos[i]); let x2 = cx + r * Math.sin(ángulos[i+1]);

```



```

let y2 = cy - r * Math. cos(ángulos[i+1]);

// Esta es una bandera para ángulos mayores que un medio círculo
// Es requerido por el componente de dibujo de arco SVG let big = (ángulos[i+1] -
ángulos[i] > Math. PI) ? 1 : 0;

// Esta cadena describe cómo dibujar una porción del gráfico circular:
let path = `M${cx},${cy}` + // Mover al centro del círculo. `L${x1},${y1}` +
// Dibuja una línea hasta (x1,y1).
`A${r},${r} 0 ${big} 1` + // Dibuja un arco de radio r... `${x2},${y2}` + //
...terminando en a (x2,y2). "Z"; // Cierra el camino de vuelta a (cx,cy).

// Calcula el color CSS para esta rodaja. Esta fórmula sólo funciona para // unos 15
colores. Así que no incluya más de 15 cortes en un gráfico. let color =
`hsl(${(i*40)%360}, ${90-3*i}%, ${50+2*i}%)`;

// Describimos un slice con un elemento <path>. Observa createElementNS().
let slice = document.createElementNS(svg, "path");

// Ahora establece los atributos en el elemento <path> slice. setAttribute("d", path); //
Establece la ruta para este slice slice. setAttribute("fill", color); // Establece el color del slice
slice. setAttribute("stroke", "black"); // Contornea el slice en negro slice.
setAttribute("stroke-width", "1");// 1 CSS pixel de grosor del gráfico.append(slice); // Añade
el slice al gráfico

// Ahora dibuja un pequeño cuadrado para la llave let icon = document.
createElementNS(svg, "rect"); icon. setAttribute("x", lx); // Coloca el cuadrado

```

```

icon.setAttribute("y", ly + 30*i); icon.setAttribute("width", 20); // Tamaño del
ícono cuadrado. setAttribute("height", 20); icon.setAttribute("fill", color); // Mismo
color de relleno que el ícono de la rebanada. setAttribute("stroke", "black"); // Mismo
contorno, también.
icon.setAttribute("stroke-width", "1"); chart.append(icon); // Añadir al gráfico

// Y añade una etiqueta a la derecha del rectángulo let label = document.
createElementNS(svg, "text"); label.setAttribute("x", lx + 30); // Posiciona el texto label.
setAttribute("y", ly + 30*i + 16); label.append(`${labels[i]} ${value}`); // Añade el texto a la
etiqueta chart.append(label); // Añade la etiqueta al gráfico });

tabla de retorno;
}

```

El gráfico de tarta de la [Figura 15-6](#) se creó utilizando la función pieChart() del [Ejemplo 15-4](#), de esta manera:

```

document.querySelector("#chart").append(pieChart({ width: 640, height: 400, //
Tamaño total del gráfico cx: 200, cy: 200, r: 180, // Centro y radio de la tarta lx: 400, ly:
10, // Posición de la leyenda data: { // Los datos a graficar "JavaScript": 71.5,
"Java": 45.4,
"Bash/Shell": 40.4,
"Python": 37.9,
"C#": 35.3,
"PHP": 31.4,
"C++": 24.6,
"C": 22.1,
"TypeScript": 18.3,
"Ruby": 10.3,

```

```
        "Swift": 8.3,  
        "Objective-C": 7.3,  
        "Ir": 7.2,  
    }  
});
```

## 15.8 Gráficos en un <lienzo>

El elemento <canvas> no tiene apariencia propia, sino que crea una superficie de dibujo dentro del documento y expone una potente API de dibujo al JavaScript del lado del cliente. La principal diferencia entre la API de <canvas> y SVG es que con el lienzo se crean dibujos llamando a métodos, y con SVG se crean dibujos construyendo un árbol de elementos XML. Estos dos enfoques son igualmente potentes: cualquiera de ellos puede simularse con el otro. Sin embargo, a primera vista son bastante diferentes y cada uno tiene sus puntos fuertes y débiles. Un dibujo SVG, por ejemplo, se edita fácilmente eliminando elementos de su descripción. Para eliminar un elemento del mismo gráfico en un <canvas>, a menudo es necesario borrar el dibujo y volver a dibujarlo desde cero. Dado que la API de dibujo de Canvas está basada en JavaScript y es relativamente compacta (a diferencia de la gramática SVG), se documenta con más detalle en este libro.

### GRÁFICOS 3D EN UN LIENZO

También puedes llamar a getContext() con la cadena "webgl" para obtener un objeto de contexto que te permita dibujar gráficos 3D utilizando la API WebGL. WebGL es una API grande, complicada y de bajo nivel que permite a los programadores de JavaScript acceder a la GPU, escribir sombreadores personalizados y realizar otras acciones muy potentes

operaciones gráficas. Sin embargo, WebGL no está documentado en este libro: los desarrolladores web son más propensos a utilizar bibliotecas de utilidades construidas sobre WebGL que utilizar la API de WebGL directamente.

La mayor parte de la API de dibujo del lienzo no está definida en el propio elemento <canvas>, sino en un objeto "contexto de dibujo" obtenido con el método getContext() del lienzo. Llame a getContext() con el argumento "2d" para obtener un objeto CanvasRenderingContext2D que puede utilizar para dibujar gráficos bidimensionales en el lienzo.

Como ejemplo sencillo de la API Canvas, el siguiente documento HTML utiliza elementos <canvas> y algo de JavaScript para mostrar dos formas sencillas:

```
<p>Esto es un cuadrado rojo: <canvas id="square" width=10  
height=10></canvas>. <p>Esto es un círculo azul: <canvas id="circle"  
width=10 height=10></canvas>. <script>  
let canvas = document.querySelector("#square"); // Obtener el primer elemento del  
lienzo let context = canvas.getContext("2d"); // Obtener el contexto de dibujo 2D  
context.fillStyle = "#f00"; // Establecer el color de relleno en rojo context.  
fillRect(0,0,10,10); // Rellenar un cuadrado  
  
canvas = document.querySelector("#circle"); // Segundo elemento del canvas context  
= canvas.getContext("2d"); // Obtiene su contexto context.beginPath(); // Inicia un  
nuevo contexto "path". arc(5, 5, 5, 0, 2*Math.PI, true); // Añade un círculo al contexto  
path. fillStyle = "#00f"; // Establece el color de relleno azul  
  
context.fill(); // Rellenar la ruta </script>
```

Hemos visto que SVG describe las formas complejas como un "camino" de líneas y curvas que se pueden dibujar o llenar. La API de Canvas también utiliza la noción de camino. En lugar de describir un camino como una cadena de letras y números, un camino se define mediante una serie de llamadas a métodos, como las invocaciones a `beginPath()` y `arc()` en el código anterior. Una vez que se define un camino, otros métodos, como `fill()`, operan sobre ese camino. Varias propiedades del objeto contexto, como `fillStyle`, especifican cómo se realizan estas operaciones.

Las subsecciones que siguen demuestran los métodos y propiedades de la API del lienzo 2D. Gran parte del código de ejemplo que sigue opera sobre una variable `c`. Esta variable contiene el objeto `CanvasRenderingContext2D` del lienzo, pero el código para inicializar esa variable a veces no se muestra. Para hacer que estos ejemplos se ejecuten, tendrías que añadir un marcado HTML para definir un lienzo con los atributos de anchura y altura apropiados, y luego añadir código como este para inicializar la variable `c`:

```
let canvas = document.querySelector("#my_canvas_id"); let c = canvas.getContext('2d');
```

### 15.8.1 Caminos y polígonos

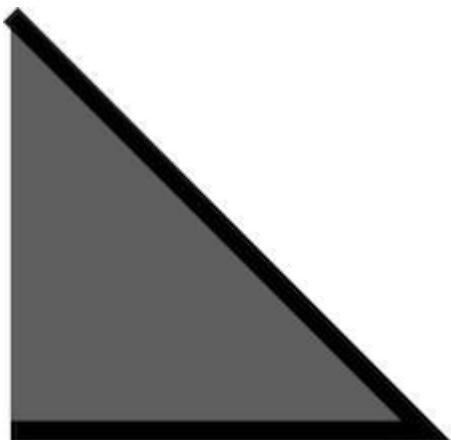
Para dibujar líneas en un lienzo y llenar las áreas delimitadas por esas líneas, se empieza por definir un *trazado*. Un trazado es una secuencia de uno o más subtrazados. Un subtrazado es una secuencia de dos o más puntos conectados por segmentos de línea (o, como veremos más adelante, por segmentos de curva). Inicie un nuevo camino con el método `beginPath()`. Inicie un nuevo subcamino con el método `moveTo()`. Una vez que haya establecido el punto de partida de un subtrayecto con `moveTo()`, puede conectar ese punto a un nuevo punto con una línea recta llamando a `lineTo()`. El siguiente código define un camino que incluye dos segmentos de línea:

- c. `beginPath(); // Iniciar una nueva ruta`
- c. `moveTo(100, 100); // Comienza un subcamino en (100,100)`
- c. `lineTo(200, 200); // Añadir una línea desde (100,100) hasta (200,200)`
- c. `lineTo(100, 200); // Añadir una línea desde (200,200) hasta (100,200)`

Este código simplemente define un camino; no dibuja nada en el lienzo. Para dibujar (o "trazar") los dos segmentos de línea en el camino, llame al método `stroke()`, y para llenar el área definida por esos segmentos de línea, llame a `fill()`:

```
c. fill(); // Rellenar un área triangular  
c. stroke(); // Trazar dos lados del triángulo
```

Este código (junto con algún código adicional para establecer los anchos de línea y los colores de relleno) produjo el dibujo mostrado en la [Figura 15-7](#).



*Figura 15-7. Una trayectoria simple, rellenada y trazada*

Observe que el subcamino definido en [la Figura 15-7](#) es "abierto". Consta de sólo dos segmentos de línea, y el punto final no está conectado al punto inicial. Esto significa que no encierra una región. El método `fill()` rellena los subcaminos abiertos actuando como si una línea recta conectara el último punto del subcamino con el primer punto del subcamino. Por eso este código rellena un triángulo, pero acaricia sólo dos lados del triángulo.

Si quieres trazar los tres lados del triángulo que acabas de mostrar, llamarías al método `closePath()` para conectar el punto final del subcamino con el punto inicial. (También podría llamar a `lineTo(100,100)`, pero entonces terminaría con tres segmentos de línea que comparten un punto de inicio y un punto final pero que no están realmente cerrados. Cuando se dibuja con líneas anchas, los resultados visuales son mejores si se utiliza `closePath()`).

Hay otros dos puntos importantes a tener en cuenta sobre `stroke()` y `fill()`. En primer lugar, ambos métodos operan en todos los subcaminos de la ruta actual. Supongamos que hemos añadido otro subcamino en el código anterior:

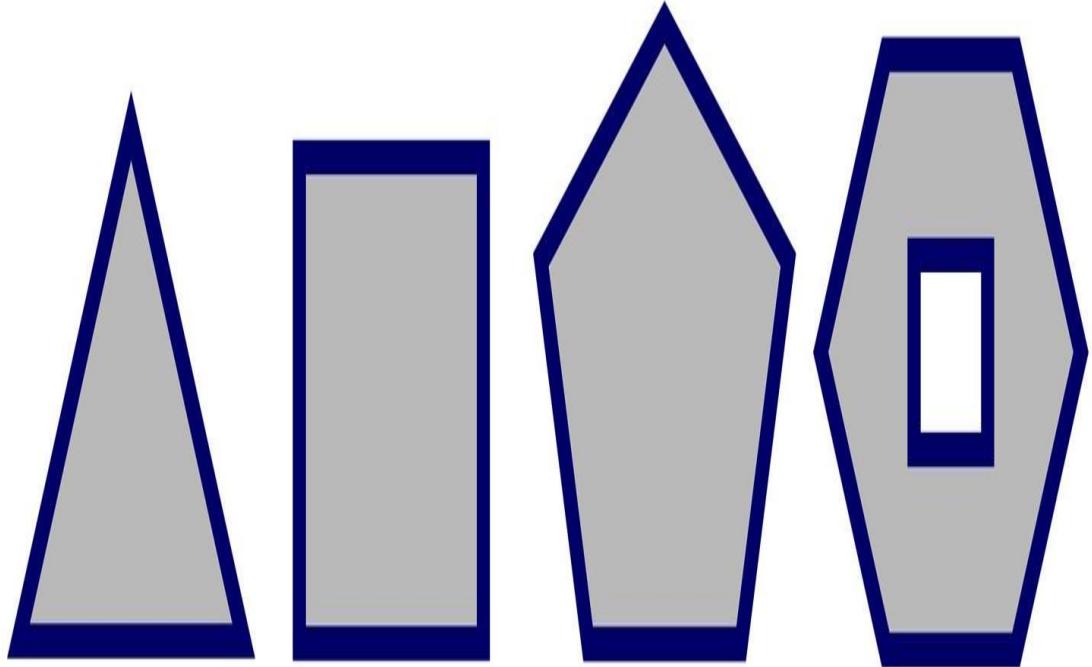
```
c. moveTo(300,100); // Comienza un nuevo subcamino en (300,100);
c. lineTo(300,200); // Dibuja una línea vertical hasta (300,200);
```

Si entonces llamáramos a `stroke()`, dibujaríamos dos aristas conectadas de un triángulo y una línea vertical desconectada.

El segundo punto a tener en cuenta sobre `stroke()` y `fill()` es que ninguno de ellos altera el trazado actual: puede llamar a `fill()` y el trazado seguirá estando ahí cuando llame a `stroke()`. Cuando haya terminado con un trazado y quiera comenzar otro, debe recordar llamar a `beginPath()`. Si no lo hace, acabará añadiendo nuevos subtrayectos al trazado existente, y puede acabar dibujando esos antiguos subtrayectos una y otra vez.

[El Ejemplo 15-5](#) define una función para dibujar polígonos regulares y demuestra el uso de `moveTo()`, `lineTo()`, y `closePath()` para definir

subtrayectos y de fill() y stroke() para dibujar esos trayectos. Produce el dibujo mostrado en la [Figura 15-8](#).



*Figura 15-8. Polígonos regulares*

*Ejemplo 15-5. Polígonos regulares con moveTo(), lineTo() y closePath()*

---

```
// Definir un polígono regular con n lados, centrado en (x,y) con radio r. // Los vértices  
están igualmente espaciados a lo largo de la circunferencia de un círculo. // Poner el  
primer vértice recto o en el ángulo especificado.
```

```
// Gira en el sentido de las agujas del reloj, a menos que el último argumento sea  
verdadero.
```

```
función polígono(c, n, x, y, r, ángulo=0,
```

```

antihorario=false) {
    c. moveTo(x + r*Math. sin(angle), // Comienza un nuevo subcamino en el primer vértice
              y - r*Math. cos(angle)); // Utilizar la trigonometría para calcular la posición
    let delta = 2*Math. PI/n; // Distancia angular entre vértices
    for(let i = 1; i < n; i++) { // Para cada uno de los vértices restantes
        ángulo += sentido contrario a las agujas del reloj?-delta:delta; // Ajustar el ángulo c.
        lineTo(x + r*Math. sin(ángulo), // Añadir línea al siguiente vértice y - r*Math. cos(ángulo));
    }
    c. closePath(); // Conectar el último vértice con el primero }

// Asumir que sólo hay un lienzo, y obtener su objeto de contexto para dibujar. let c =
document. querySelector("canvas"). getContext("2d");

// Iniciar un nuevo camino y añadir subcaminos poligonales
c. beginPath(); polygon(c, 3, 50, 70, 50); // Triángulo polygon(c, 4, 150, 60, 50, Math.
PI/4); // Cuadrado polygon(c, 5, 255, 55, 50); // Pentágono polygon(c, 6, 365, 53, 50,
Math. PI/6); // Hexágono polygon(c, 4, 365, 53, 20, Math. PI/4, true); // Pequeño
cuadrado dentro del hexágono

// Establece algunas propiedades que controlan el aspecto de los gráficos
c. fillStyle = "#ccc"; // Interiores gris claro
c. strokeStyle = "#008"; // perfilado con líneas azul oscuro
c. lineWidth = 5; // cinco píxeles de ancho.

// Ahora dibuja todos los polígonos (cada uno en su propia sub-ruta) con estas llamadas
c. fill(); // Rellenar las formas
c. stroke(); // Y trazar sus contornos

```

Observe que este ejemplo dibuja un hexágono con un cuadrado en su interior. El cuadrado y el hexágono son subtrayectos separados, pero se superponen. Cuando esto ocurre (o cuando un subcamino se cruza a sí mismo), el lienzo necesita ser capaz de determinar qué regiones están dentro del camino y cuáles están fuera. Para ello, el lienzo utiliza una prueba conocida como "regla de enrollamiento no nulo". En este caso, el interior del cuadrado no se rellena porque el cuadrado y el hexágono se dibujaron en direcciones opuestas: los vértices del hexágono se conectaron con segmentos de línea que se mueven en el sentido de las agujas del reloj alrededor del círculo. Los vértices del cuadrado se conectaron en sentido contrario a las agujas del reloj. Si el cuadrado se hubiera dibujado también en el sentido de las agujas del reloj, la llamada a `fill()` habría llenado también el interior del cuadrado.

### 15.8.2 Dimensiones y coordenadas del lienzo

Los atributos de anchura y altura del elemento `<canvas>` y las correspondientes propiedades de anchura y altura del objeto `Canvas` especifican las dimensiones del lienzo. El sistema de coordenadas del lienzo por defecto sitúa el origen `(0,0)` en la esquina superior izquierda del lienzo. Las coordenadas `x` *aumentan* hacia la derecha y las coordenadas `y` aumentan a medida que se desciende en la pantalla. Los puntos del lienzo pueden especificarse utilizando valores de punto flotante.

Las dimensiones de un lienzo no pueden ser alteradas sin restablecer completamente el lienzo. Al establecer las propiedades de anchura o altura de un lienzo (incluso fijándolas en su valor actual) se borra el

lienzo, se borra el trazado actual y se restablecen todos los atributos gráficos (incluyendo la transformación actual y la región de recorte) a su estado original.

Los atributos de anchura y altura de un lienzo especifican el número real de píxeles que el lienzo puede dibujar. Se asignan cuatro bytes de memoria para cada píxel, por lo que si la anchura y la altura se establecen en 100, el lienzo asigna 40.000 bytes para representar 10.000 píxeles.

Los atributos de anchura y altura también especifican el tamaño por defecto (en píxeles CSS) al que se mostrará el lienzo en la pantalla. Si `window.devicePixelRatio` es 2, entonces  $100 \times 100$  píxeles CSS son en realidad 40.000 píxeles de hardware. Cuando el contenido del lienzo se dibuja en la pantalla, los 10.000 píxeles de la memoria tendrán que ampliarse para cubrir 40.000 píxeles físicos en la pantalla, y esto significa que sus gráficos no serán tan nítidos como podrían ser.

Para obtener una calidad de imagen óptima, no debe utilizar los atributos de anchura y altura para establecer el tamaño en pantalla del lienzo. En su lugar, establezca el tamaño de píxel CSS deseado en pantalla del lienzo con los atributos de estilo CSS `width` y `height`. A continuación, antes de empezar a dibujar en su código JavaScript, establezca las propiedades de anchura y altura del objeto canvas con el número de píxeles CSS multiplicado por `window.devicePixelRatio`. Siguiendo con lo anterior

Por ejemplo, esta técnica haría que el lienzo se mostrara a  $100 \times 100$  píxeles CSS pero asignando memoria para  $200 \times 200$  píxeles. (Incluso

con esta técnica, el usuario puede ampliar el lienzo y, si lo hace, puede ver gráficos borrosos o pixelados. Esto contrasta con los gráficos SVG, que permanecen nítidos sin importar el tamaño en pantalla o el nivel de zoom).

### 15.8.3 Atributos de los gráficos

El Ejemplo 15-5 establece las propiedades `fillStyle`, `strokeStyle` y `lineWidth` en el objeto contexto del lienzo. Estas propiedades son atributos gráficos que especifican el color a utilizar por `fill()` y por `stroke()`, y el ancho de las líneas a dibujar por `stroke()`. Observe que estos parámetros no se pasan a los métodos `fill()` y `stroke()`, sino que forman parte del *estado gráfico* general del lienzo. Si defines un método que dibuja una forma y no estableces estas propiedades, el que llama a tu método puede definir el color de la forma estableciendo las propiedades `strokeStyle` y `fillStyle` antes de llamar a tu método. Esta separación entre el estado de los gráficos y los comandos de dibujo es fundamental para la API de Canvas y es similar a la separación entre la presentación y el contenido que se logra al aplicar hojas de estilo CSS a los documentos HTML.

Hay una serie de propiedades (y también algunos métodos) en el objeto contexto que afectan al estado gráfico del lienzo. Se detallan a continuación.

#### ESTILOS DE LÍNEA

La propiedad `lineWidth` especifica el ancho (en píxeles CSS) que tendrán las líneas dibujadas por `stroke()`. El valor por defecto es 1. Es

importante entender que el ancho de la línea se determina por la propiedad `lineWidth` en el momento en que se llama a `stroke()`, no en el momento en que se llama a `lineTo()` y a otros métodos de construcción de trazados. Para entender completamente la propiedad `lineWidth`, es importante visualizar los trazados como líneas unidimensionales infinitamente finas. Las líneas y curvas dibujadas por el método `stroke()` están centradas sobre el trazado, con la mitad del `lineWidth` a cada lado. Si está trazando un trazado cerrado y sólo quiere que la línea aparezca fuera del trazado, trace primero el trazado y luego rellene con un color opaco para ocultar la parte del trazado que aparece dentro del trazado. O si sólo quiere que la línea aparezca dentro de un trazado cerrado, llame primero a los métodos `save()` y `clip()`, y luego llame a `stroke()` y `restore()`. (Los métodos `save()`, `restore()` y `clip()` se describen más adelante).

Cuando se dibujan líneas de más de dos píxeles de ancho, las propiedades `lineCap` y `lineJoin` pueden tener un impacto significativo en la apariencia visual de los extremos de un trazado y de los vértices en los que se encuentran dos segmentos del trazado. [La Figura 15-9](#) ilustra los valores y el aspecto gráfico resultante de `lineCap` y `lineJoin`.

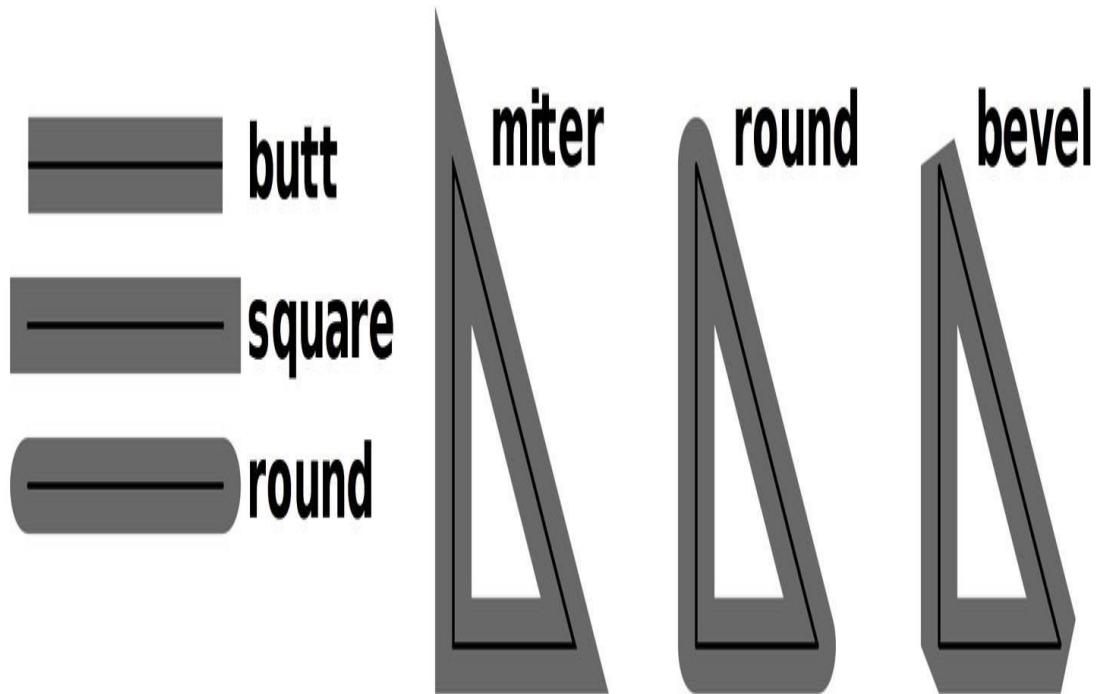


Figura 15-9. Los atributos `lineCap` y `lineJoin`

El valor por defecto de `lineCap` es "butt". El valor por defecto de `lineJoin` es "inglete". Tenga en cuenta, sin embargo, que si dos líneas se unen en un ángulo muy estrecho, entonces el inglete resultante puede llegar a ser bastante largo y distraer visualmente. Si el inglete en un vértice dado fuera más largo que la mitad del ancho de la línea multiplicado por la propiedad `miterLimit`, ese vértice se dibujará con una unión en bisel en lugar de una unión en inglete. El valor por defecto de `miterLimit` es 10.

El método `stroke()` puede dibujar líneas discontinuas y punteadas, así como líneas sólidas, y el estado gráfico de un lienzo incluye una matriz de números que sirve como "patrón de guiones" especificando cuántos píxeles dibujar, y luego cuántos omitir. A diferencia de otras propiedades de dibujo de líneas, el patrón de guiones se establece y

consulta con los métodos `setLineDash()` y `getLineDash()` en lugar de con una propiedad. Para especificar un patrón de guiones, puede utilizar `setLineDash()` así:

```
c. setLineDash([18, 3, 3, 3]); // 18px guión, 3px espacio, 3px punto, 3px espacio
```

Finalmente, la propiedad `lineDashOffset` especifica a qué distancia debe comenzar el dibujo del patrón de guiones. El valor predeterminado es 0. Las rutas trazadas con el patrón de guiones que se muestra aquí comienzan con un guión de 18 píxeles, pero si `lineDashOffset` se establece en 21, esa misma ruta comenzaría con un punto seguido de un espacio y un guión.

## COLORES, PATRONES Y DEGRADADOS

Las propiedades `fillStyle` y `strokeStyle` especifican cómo se rellenan y trazan los trazados. La palabra "estilo" suele significar color, pero estas propiedades también pueden utilizarse para especificar un gradiente de color o una imagen que se utilizará para el relleno y el trazo. (Tenga en cuenta que dibujar una línea es básicamente lo mismo que llenar una región estrecha a ambos lados de la línea, y el relleno y el trazo son fundamentalmente la misma operación).

Si desea llenar o trazar con un color sólido (o un color translúcido), simplemente establezca estas propiedades a una cadena de color CSS válida. No se requiere nada más.

Para llenar (o trazar) con un gradiente de color, establezca `fillStyle` (o `strokeStyle`) a un objeto `CanvasGradient` devuelto por la función

createLinearGradient() o createRadialGradient()  
del contexto. Los argumentos para  
createLinearGradient() son las coordenadas de dos puntos que definen  
una línea (no es necesario que sea horizontal o vertical) a lo largo de la  
cual variarán los colores. Los argumentos de createRadialGradient()  
especifican los centros y radios de dos círculos. (No es necesario que  
sean concéntricos, pero el primer círculo suele estar completamente  
dentro del segundo). Las áreas dentro del círculo más pequeño o fuera  
del más grande se llenarán con colores sólidos; las áreas entre los  
dos se llenarán con un gradiente de color.

Después de crear el objeto CanvasGradient que define las regiones del  
lienzo que serán llenadas, debe definir los colores del gradiente  
llamando al método addColorStop() del CanvasGradient. El primer  
argumento de este método es un número entre 0.0 y 1.0. El segundo  
argumento es una especificación de color CSS. Debe llamar a este  
método al menos dos veces para definir un gradiente de color simple,  
pero puede llamarlo más veces. El color en 0.0 aparecerá al principio  
del gradiente, y el color en 1.0 aparecerá al final. Si especifica colores  
 adicionales, aparecerán en la posición fraccional especificada dentro  
 del gradiente.

Entre los puntos que especifique, los colores se interpolarán  
suavemente.

He aquí algunos ejemplos:

```
// Un gradiente lineal, en diagonal a través del lienzo (asumiendo que no hay  
transformaciones) let bgfade =
```

```

c.createLinearGradient(0,0,canvas.width,canvas.height); bgfade.addColorStop(0.0,
"#88f"); // Empezar con azul claro en la parte superior izquierda bgfade.
addColorStop(1.0, "#fff"); // Desvanecerse a blanco en la parte inferior derecha

// Un gradiente entre dos círculos concéntricos. Transparente en el centro //
desvaneciéndose a gris translúcido y luego de vuelta a transparente. let donut = c.
createRadialGradient(300,300,100, 300,300,300); donut.addColorStop(0.0,
"transparent"); // Transparente
donut.addColorStop(0.7, "rgba(100,100,100,.9)"); // Gris translúcido
donut.addColorStop(1.0, "rgba(0,0,0,0)"); // Transparente de nuevo

```

Un punto importante a entender sobre los gradientes es que no son independientes de la posición. Cuando se crea un degradado, se especifican los límites del mismo. Si luego intenta llenar un área fuera de esos límites, obtendrá el color sólido definido en un extremo u otro del degradado.

Además de los colores y los degradados de color, también puede llenar y trazar utilizando imágenes. Para ello, establezca `fillStyle` o `strokeStyle` a un `CanvasPattern` devuelto por el método `createPattern()` del objeto `context`. El primer argumento de este método debe ser un elemento `<img>` o `<canvas>` que contenga la imagen con la que se quiere llenar o trazar. (Tenga en cuenta que no es necesario que la imagen o el lienzo de origen estén insertados en el documento para poder utilizarlos de este modo). El segundo argumento de `createPattern()` es la cadena "repeat", "repeat-x", "repeat-y" o "no-repeat", que especifica si las imágenes de fondo se repiten (y en qué dimensiones).

## ESTILOS DE TEXTO

La propiedad font especifica el tipo de letra a utilizar por los métodos de dibujo de texto `fillText()` y `strokeText()` (ver "[Texto](#)"). El valor de la propiedad font debe ser una cadena con la misma sintaxis que el atributo font de CSS.

La propiedad `textAlign` especifica cómo debe alinearse el texto horizontalmente con respecto a la coordenada X pasada a `fillText()` o `strokeText()`. Los valores legales son "inicio", "izquierda", "centro", "derecha" y "final". El valor por defecto es "inicio", que, para el texto de izquierda a derecha, tiene el mismo significado que "izquierda".

La propiedad `textBaseline` especifica cómo debe alinearse el texto verticalmente con respecto a la coordenada y. El valor por defecto es "alfabético", y es apropiado para escrituras latinas y similares. El valor "ideográfico" está pensado para usar con escrituras como la china y la japonesa. El valor "colgante" está pensado para utilizarlo con el devanagari y escrituras similares (que se utilizan para muchas de las lenguas de la India). Las líneas de base "superior", "central" e "inferior" son líneas de base puramente geométricas, basadas en el "cuadrado em" de la fuente.

## SOMBRA

Cuatro propiedades del objeto contextual controlan el dibujo de las sombras. Si establece estas propiedades adecuadamente, cualquier

línea, área, texto o imagen que dibuje recibirá una sombra, que hará que parezca que está flotando sobre la superficie del lienzo.

La propiedad shadowColor especifica el color de la sombra. El valor por defecto es el negro totalmente transparente, y las sombras nunca aparecerán a menos que se establezca esta propiedad a un color translúcido u opaco. Esta propiedad sólo puede establecerse a una cadena de color: los patrones y los gradientes no están permitidos para las sombras. El uso de un color de sombra translúcido produce los efectos de sombra más realistas porque permite que se vea el fondo.

Las propiedades shadowOffsetX y shadowOffsetY especifican el Desplazamientos X e Y de la sombra. El valor por defecto de ambas propiedades es 0, lo que coloca la sombra directamente debajo de su dibujo, donde no es visible. Si establece ambas propiedades en un valor positivo, las sombras aparecerán por debajo y a la derecha de lo que dibuje, como si hubiera una fuente de luz por encima y a la izquierda, iluminando el lienzo desde fuera de la pantalla del ordenador. Los desplazamientos más grandes producen sombras más grandes y hacen que los objetos dibujados aparezcan como si estuvieran flotando "más alto" sobre el lienzo. Estos valores no se ven afectados por las transformaciones de coordenadas ([§15.8.5](#)): la dirección y la "altura" de la sombra permanecen constantes incluso cuando las formas se rotan y escalan.

La propiedad shadowBlur especifica el grado de desenfoque de los bordes de la sombra. El valor por defecto es 0, que produce sombras nítidas y no borrosas. Los valores más grandes producen

más desenfoque, hasta un límite superior definido por la implementación.

## TRANSLUCIDEZ Y COMPOSICIÓN

Si quieras trazar o rellenar un trazado usando un color translúcido, puedes establecer `strokeStyle` o `fillStyle` usando una sintaxis de color CSS como `"rgba(...)"` que soporte la transparencia alfa. La "a" en "RGBA" significa "alfa" y es un valor entre 0 (totalmente transparente) y 1 (totalmente opaco). Pero la API del lienzo ofrece otra forma de trabajar con colores translúcidos. Si no quieras especificar explícitamente un canal alfa para cada color, o si quieres añadir translucidez a imágenes o patrones opacos, puedes establecer la propiedad `globalAlpha`. Cada píxel que dibujes tendrá su valor alfa multiplicado por `globalAlpha`. El valor por defecto es 1, que no añade transparencia. Si estableces `globalAlpha` a 0, todo lo que dibujes será totalmente transparente, y no aparecerá nada en el lienzo. Pero si estableces esta propiedad a 0.5, entonces los píxeles que hubieran sido opacos serán 50% opacos, y los píxeles que hubieran sido 50% opacos serán 25% opacos en su lugar.

Cuando se trazan líneas, se rellenan regiones, se dibuja texto o se copian imágenes, generalmente se espera que los nuevos píxeles se dibujen encima de los píxeles que ya están en el lienzo. Si estás dibujando píxeles opacos, simplemente reemplazan a los píxeles que ya están ahí. Si está dibujando con píxeles translúcidos, el nuevo píxel ("fuente") se combina con el antiguo píxel ("destino") de manera que el antiguo píxel se muestra a través del nuevo píxel en función de lo transparente que sea ese píxel.

Este proceso de combinar nuevos píxeles de origen (posiblemente translúcidos) con píxeles de destino existentes (posiblemente translúcidos) se denomina *composición*, y el proceso de composición descrito anteriormente es la forma predeterminada en que la API de Canvas combina los píxeles. Pero puede establecer la propiedad globalCompositeOperation para especificar otras formas de combinar píxeles. El valor por defecto es "source-over", lo que significa que los píxeles de origen se dibujan "sobre" los píxeles de destino y se combinan con ellos si el origen es translúcido. Pero si establece globalCompositeOperation como "destination-over", entonces el lienzo combinará los píxeles como si los nuevos píxeles de origen se dibujaran debajo de los píxeles de destino existentes. Si el destino es translúcido o transparente, parte o todo el color del píxel de origen es visible en el color resultante. Como otro ejemplo, el modo de composición "sourceatop" combina los píxeles de origen con la transparencia de los píxeles de destino para que no se dibuje nada en las partes del lienzo que ya son totalmente transparentes. Hay un número de valores legales para globalCompositeOperation, pero la mayoría tienen sólo usos especializados y no se cubren aquí.

## GUARDAR Y RESTAURAR EL ESTADO DE LOS GRÁFICOS

Dado que la API de Canvas define los atributos de los gráficos en el objeto contexto, podría estar tentado de llamar a getContext() varias veces para obtener varios objetos contexto. Si pudiera hacer esto, podría definir diferentes atributos en cada contexto: cada contexto sería entonces como un pincel diferente y pintaría con un color diferente o dibujaría líneas de diferente anchura. Lamentablemente,

no se puede utilizar el lienzo de esta manera. Cada elemento <canvas> tiene un solo objeto contexto, y cada llamada a getContext() devuelve el mismo objeto CanvasRenderingContext2D.

Aunque la API del lienzo sólo permite definir un único conjunto de atributos gráficos a la vez, permite guardar el estado actual de los gráficos para poder modificarlo y restaurarlo fácilmente más tarde. El método save() coloca el estado actual de los gráficos en una pila de estados guardados. El método restore() abre la pila y restaura el último estado guardado. Todas las propiedades que se han descrito en esta sección forman parte del estado guardado, así como la transformación actual y la región de recorte (ambas se explican más adelante). Es importante destacar que la trayectoria definida actualmente y el punto actual no forman parte del estado de los gráficos y no pueden ser guardados y restaurados.

#### **15.8.4 Operaciones de dibujo en el lienzo**

Ya hemos visto algunos métodos básicos del lienzo: beginPath(), moveTo(), lineTo(), closePath(), fill() y stroke() -para definir, rellenar y dibujar líneas y polígonos. Pero la API de Canvas también incluye otros métodos de dibujo.

#### **RECTÁNGULOS**

CanvasRenderingContext2D define cuatro métodos para dibujar rectángulos. Los cuatro métodos de rectángulo esperan dos argumentos que especifican una esquina del rectángulo seguida de la anchura y la altura del mismo. Normalmente, se especifica la esquina

superior izquierda y se pasa una anchura y una altura positivas, pero también se pueden especificar otras esquinas y pasar dimensiones negativas.

`fillRect()` rellena el rectángulo especificado con el `fillStyle` actual. `strokeRect()` traza el contorno del rectángulo especificado utilizando el `strokeStyle` actual y otros atributos de línea. `clearRect()` es como `fillRect()`, pero ignora el estilo de relleno actual y rellena el rectángulo con píxeles negros transparentes (el color por defecto de todos los lienzos en blanco). Lo importante de estos tres métodos es que no afectan al trazado actual ni al punto actual dentro de ese trazado.

El último método de rectángulo se llama `rect()`, y afecta al trazado actual: añade el rectángulo especificado, en un subtrazado propio, al trazado. Al igual que otros métodos de definición de trayectorias, no rellena ni traza nada por sí mismo.

## CURVAS

Un camino es una secuencia de subcaminos, y un subcamino es una secuencia de puntos conectados. En los trayectos que definimos en §15.8.1, esos puntos estaban conectados con segmentos de líneas rectas, pero eso no tiene por qué ser siempre así. El objeto `CanvasRenderingContext2D` define una serie de métodos que añaden un nuevo punto al subtrazado y conectan el punto actual a ese nuevo punto con una curva:

*arco()*

Este método añade un círculo, o una parte de un círculo (un arco), a la trayectoria. El arco a dibujar se especifica con seis parámetros: las coordenadas x e y del centro del círculo, el radio del círculo, los ángulos inicial y final del arco y la dirección (en el sentido de las agujas del reloj o en sentido contrario) del arco entre esos dos ángulos. Si hay un punto actual en la trayectoria, este método conecta el punto actual con el inicio del arco con una línea recta (lo que es útil cuando se dibujan cuñas o rebanadas de pastel), luego conecta el inicio del arco con el final del arco con una porción de círculo, dejando el final del arco como el nuevo punto actual. Si no hay ningún punto actual cuando se llama a este método, entonces sólo añade el arco circular a la trayectoria.

### *elipse()*

Este método es muy parecido a arc(), excepto que añade una elipse o una parte de una elipse a la trayectoria. En lugar de un radio, tiene dos: un radio *en el eje x* y un radio *en el eje y*. Además, como las elipses no son radialmente simétricas, este método toma otro argumento que especifica el número de radianes en los que la elipse se gira en el sentido de las agujas del reloj alrededor de su centro.

### *arcTo()*

Este método dibuja una línea recta y un arco circular al igual que el método arc(), pero especifica el arco a dibujar utilizando diferentes parámetros. Los argumentos de arcTo() especifican los puntos P1 y P2 y un radio. El arco que se añade a la trayectoria tiene el radio especificado. Comienza en el punto tangente con la línea (imaginaria) desde el punto actual a P1 y termina en el punto tangente con la línea (imaginaria) entre P1 y P2. Este método de especificación de arcos, que parece poco habitual, es en realidad bastante útil para dibujar formas con esquinas redondeadas. Si especifica un radio de 0, este método simplemente dibuja una línea recta desde el punto actual hasta P1. Sin embargo, con un radio distinto de cero, dibuja una línea

recta desde el punto actual en la dirección de P1, y luego curva esa línea en un círculo hasta que se dirige en la dirección de P2.

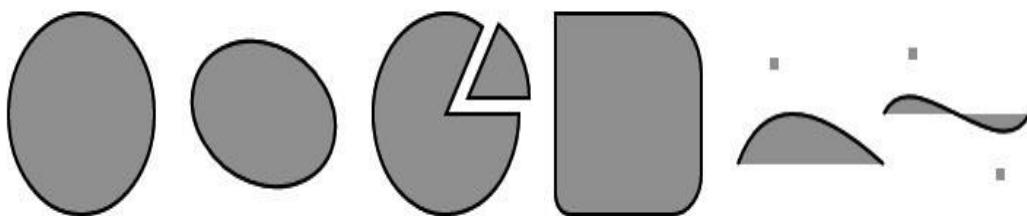
#### *bezierCurveTo()*

Este método añade un nuevo punto P a la sub-ruta y lo conecta al punto actual con una curva cúbica de Bézier. La forma de la curva se especifica mediante dos "puntos de control", C1 y C2. Al principio de la curva (en el punto actual), la curva se dirige en la dirección de C1. Al final de la curva (en el punto P), la curva llega desde la dirección de C2. Entre estos puntos, la dirección de la curva varía suavemente. El punto P se convierte en el nuevo punto actual del subcamino.

#### *quadraticCurveTo()*

Este método es como `bezierCurveTo()`, pero utiliza una curva de Bézier cuadrática en lugar de una curva de Bézier cúbica y sólo tiene un punto de control.

Puede utilizar estos métodos para dibujar trazados como los de la [Figura 15-10](#).



*Figura 15-10. Trayectos curvos en un lienzo*

[El Ejemplo 15-6](#) muestra el código utilizado para crear [la Figura 15-10](#). Los métodos demostrados en este código son algunos de los más

complicados de la API de Canvas; consulte una referencia en línea para obtener detalles completos sobre los métodos y sus argumentos.

#### *Ejemplo 15-6. Añadir curvas a una trayectoria*

---

```
// Una función de utilidad para convertir ángulos de grados a radianes function rads(x) {
return Math.PI*x/180; }

// Obtener el objeto contexto del elemento canvas del documento let c = document.
querySelector("canvas").getContext("2d");

// Definir algunos atributos gráficos y dibujar las curvas
c.fillStyle = "#aaa"; // Rellenos grises
c.lineWidth = 2; // Líneas negras de 2 píxeles (por defecto)
```

```
// Dibuja un círculo.  
// No hay ningún punto actual, así que dibuja sólo el círculo sin ninguna // línea recta  
desde el punto actual hasta el inicio del círculo. c.beginPath();  
c.arc(75,100,50, // Centro en (75,100), radio 50 0,rads(360),false); // Ir en el sentido de las  
agujas del reloj de 0 a 360 grados  
c.fill(); // Rellenar el círculo  
c.stroke(); // Trazar su contorno.  
  
// Ahora dibuja una elipse de la misma manera  
c.beginPath(); // Iniciar un nuevo camino no conectado al círculo  
c.ellipse(200, 100, 50, 35, rads(15), // Centro, radios y rotación 0, rads(360), false); //  
Ángulo inicial, ángulo final, dirección  
  
// Dibuja una cuña. Los ángulos se miden en el sentido de las agujas del reloj desde el  
eje x positivo. // Observe que arc() añade una línea desde el punto actual hasta el  
inicio del arco.  
c.moveTo(325, 100); // Comienza en el centro del círculo.  
c.arc(325, 100, 50, // Centro del círculo y radio rads(-60), rads(0), // Comienza en el ángulo  
-60 y va al ángulo 0  
true); // en sentido contrario a las agujas del reloj  
c.closePath(); // Añadir el radio al centro del círculo  
  
// Cuña similar, desplazada un poco y en sentido contrario c.moveTo(340, 92);  
c.arc(340, 92, 42, rads(-60), rads(0), false);  
c.closePath();  
  
// Utilice arcTo() para las esquinas redondeadas. Aquí dibujamos un cuadrado con //  
esquina superior izquierda en (400,50) y esquinas de radios variables.  
c.moveTo(450, 50); // Comienza en el centro del borde superior.  
c.arcTo(500,50,500,150,30); // Añadir parte del borde superior y la esquina superior  
derecha.  
c.arcTo(500,150,400,150,20); // Añadir el borde derecho y la esquina inferior derecha.
```

```

c. arcTo(400,150,400,50,10); // Añadir el borde inferior y la esquina inferior izquierda.
c. arcTo(400,50,500,50,0); // Añadir el borde izquierdo y la esquina superior izquierda.
c. closePath(); // Cerrar el camino para añadir el resto del borde superior.

// Curva de Bézier cuadrática: un punto de control
c. moveTo(525, 125); // Comienza aquí
c. quadraticCurveTo(550, 75, 625, 125); // Dibuja una curva hasta (625, 125)
c. fillRect(550-3, 75-3, 6, 6); // Marcar el punto de control (550,75)

// Curva cúbica de Bézier
c. moveTo(625, 100); // Comienza en (625, 100)
c. bezierCurveTo(645,70,705,130,725,100); // Curva a (725, 100)
c. fillRect(645-3, 70-3, 6, 6); // Marcar puntos de control
c. fillRect(705-3, 130-3, 6, 6);

// Por último, rellena las curvas y traza sus contornos.
c. fill();
c. stroke();

```

## TEXTO

Para dibujar texto en un lienzo, normalmente se utiliza el método `fillText()`, que dibuja el texto utilizando el color (o gradiente o patrón) especificado por la propiedad `fillStyle`. Para obtener efectos especiales en tamaños de texto grandes, puede utilizar `strokeText()` para dibujar el contorno de los glifos individuales de la fuente. Ambos métodos toman el texto a dibujar como primer argumento y toman las coordenadas `x` e `y` del texto como segundo y tercer argumento.

Ninguno de los dos métodos afecta al trazado actual o al punto actual.

`fillText()` y `strokeText()` toman un cuarto argumento opcional. Si se da, este argumento especifica la anchura máxima del texto a mostrar. Si el texto fuera más ancho que el valor especificado cuando se dibuja utilizando la propiedad de la fuente, el lienzo lo hará encajar escalándolo o utilizando una fuente más estrecha o más pequeña.

Si necesita medir el texto usted mismo antes de dibujarlo, páselo al método `measureText()`. Este método devuelve un objeto `TextMetrics` que especifica las medidas del texto cuando se dibuja con la fuente actual. En el momento de escribir esto, la única "métrica" contenida en el objeto `TextMetrics` es la anchura. Consulte la anchura en pantalla de una cadena así:

```
let width = c.measureText(text).width;
```

Esto es útil si quiere centrar una cadena de texto dentro de un lienzo, por ejemplo.

## IMÁGENES

Además de los gráficos vectoriales (trazados, líneas, etc.), la API del lienzo también admite imágenes de mapa de bits. El método `drawImage()` copia los píxeles de una imagen de origen (o de un rectángulo dentro de la imagen de origen) en el lienzo, escalando y rotando los píxeles de la imagen según sea necesario.

`drawImage()` puede ser invocada con tres, cinco o nueve argumentos. En todos los casos, el primer argumento es la imagen de origen de la que se van a copiar los píxeles. Este argumento de la imagen suele ser un elemento `<img>`, pero también puede ser otro elemento `<canvas>` o incluso un elemento `<video>` (del que se copiará un solo fotograma). Si se especifica un elemento `<img>` o `<video>` que todavía está cargando sus datos, la llamada a `drawImage()` no hará nada.

En la versión de tres argumentos de `drawImage()`, el segundo y tercer argumento especifican las coordenadas `x` e `y` en las que se dibujará la

esquina superior izquierda de la imagen. En esta versión del método, toda la imagen de origen se copia en el lienzo. Las coordenadas  $x$  e  $y$  se interpretan en el sistema de coordenadas actual, y la imagen se escala y rota si es necesario, dependiendo de la transformación del lienzo actualmente en efecto.

La versión de cinco argumentos de `drawImage()` añade argumentos de anchura y altura a los argumentos  $x$  e  $y$  descritos anteriormente. Estos cuatro argumentos definen un rectángulo de destino dentro del lienzo. La esquina superior izquierda de la imagen de origen va a  $(x,y)$ , y la esquina inferior derecha va a  $(x+ancho, y+alto)$ . De nuevo, se copia toda la imagen de origen. Con esta versión del método, la imagen de origen se escalará para ajustarse al rectángulo de destino.

La versión de nueve argumentos de `drawImage()` especifica un rectángulo de origen y un rectángulo de destino y copia sólo los píxeles dentro del rectángulo de origen. Los argumentos dos a cinco especifican el rectángulo de origen. Se miden en píxeles CSS. Si la imagen de origen es otro lienzo, el rectángulo de origen utiliza el sistema de coordenadas por defecto de ese lienzo e ignora cualquier transformación que se haya especificado. Los argumentos del seis al nueve especifican el rectángulo de destino en el que se dibuja la imagen y están en el sistema de coordenadas actual del lienzo, no en el sistema de coordenadas por defecto.

Además de dibujar imágenes en un lienzo, también podemos extraer el contenido de un lienzo como una imagen utilizando el método `toDataURL()`.

A diferencia de todos los demás métodos descritos aquí, `toDataURL()` es un método del propio elemento Canvas, no del objeto `context`. Normalmente se invoca a `toDataURL()` sin argumentos, y devuelve el contenido del lienzo como una imagen PNG, codificada como una cadena usando un `data:` URL. La URL devuelta es adecuada para su uso con un elemento `<img>`, y puede hacer una instantánea estática de un lienzo con un código como este:

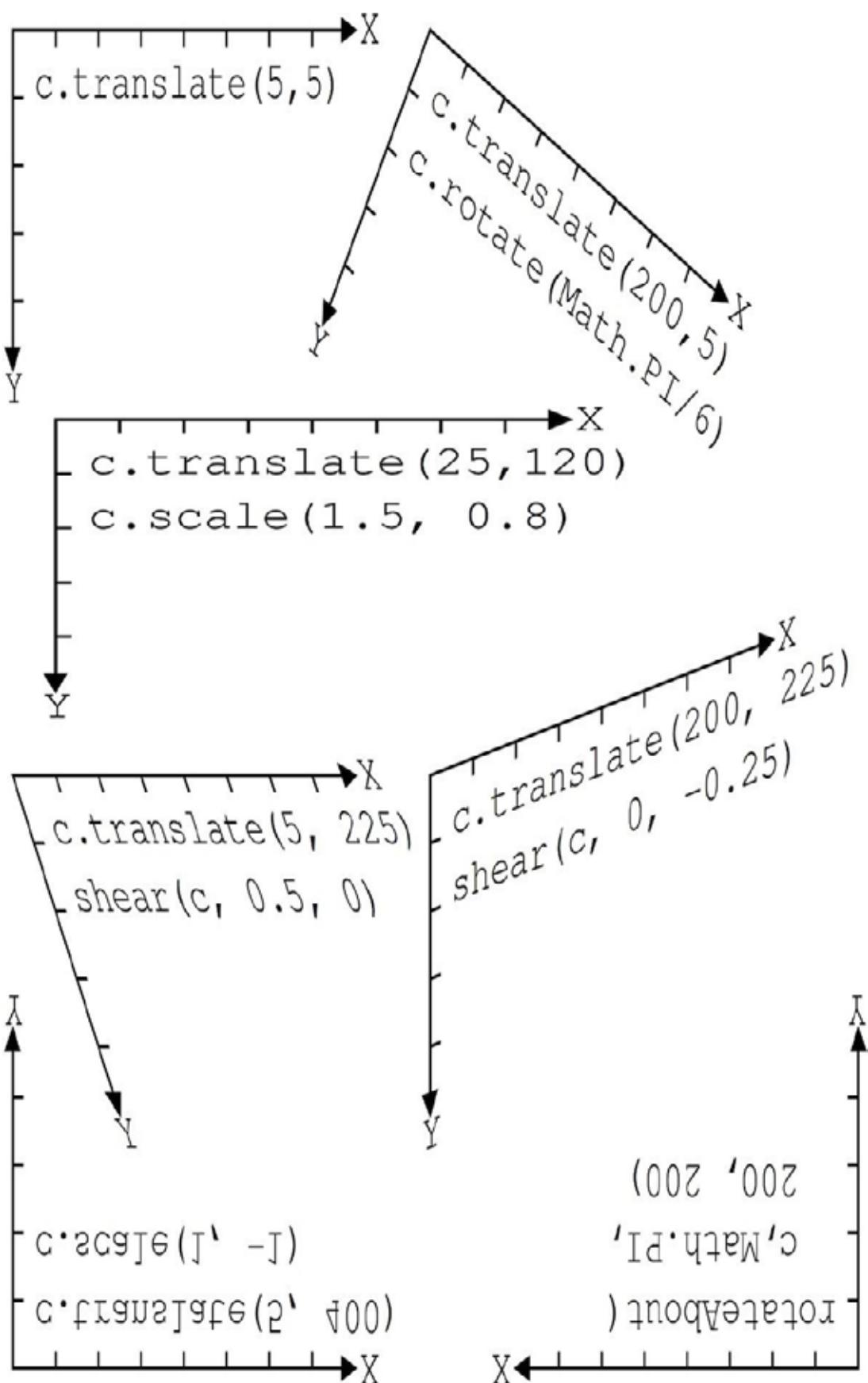
```
let img = document.createElement("img"); // Crear un elemento <img> img. src = canvas.  
toDataURL(); // Establecer su atributo src document. body. appendChild(img); // Añadirlo  
al documento
```

### 15.8.5 Transformaciones del sistema de coordenadas

Como hemos señalado, el sistema de coordenadas por defecto de un lienzo sitúa el origen en la esquina superior izquierda, tiene coordenadas `x` que aumentan hacia la derecha, y tiene coordenadas `y` que aumentan hacia abajo. En este sistema por defecto, las coordenadas de un punto se asignan directamente a un píxel CSS (que a su vez se asigna directamente a uno o más píxeles del dispositivo). Ciertas operaciones y atributos del lienzo (como la extracción de valores de píxeles sin procesar y el establecimiento de desplazamientos de sombra) siempre utilizan este sistema de coordenadas por defecto. Sin embargo, además del sistema de coordenadas por defecto, cada lienzo tiene una "matriz de transformación actual" como parte de su estado gráfico. Esta matriz define el sistema de coordenadas actual del lienzo. En la mayoría de las operaciones del lienzo, cuando se especifican las coordenadas de un punto, se toma como un punto en el sistema de coordenadas

actual, no en el sistema de coordenadas por defecto. La matriz de transformación actual se utiliza para convertir las coordenadas especificadas en las coordenadas equivalentes en el sistema de coordenadas por defecto.

El método `setTransform()` permite establecer directamente la matriz de transformación de un lienzo, pero las transformaciones del sistema de coordenadas suelen ser más fáciles de especificar como una secuencia de traslaciones, rotaciones y operaciones de escalado. La Figura 15-11 ilustra estas operaciones y su efecto en el sistema de coordenadas del lienzo. El programa que produjo la figura dibujó el mismo conjunto de ejes siete veces seguidas. Lo único que cambió cada vez fue la transformación actual. Observe que las transformaciones afectan tanto al texto como a las líneas que se dibujan.



*Figura 15-11. Transformaciones del sistema de coordenadas*

El método `translate()` simplemente mueve el origen del sistema de coordenadas a la izquierda, derecha, arriba o abajo. El método `rotate()` gira los ejes en el sentido de las agujas del reloj en el ángulo especificado. (La API de Canvas siempre especifica los ángulos en radianes. Para convertir los grados en radianes, divida por 180 y multiplicar por `Math.PI`). El método `scale()` estira o contrae las distancias a lo largo de los ejes `x` o `y`.

Pasar un factor de escala negativo al método `scale()` invierte ese eje a través del origen, como si se reflejara en un espejo. Esto es lo que se hizo en la parte inferior izquierda de la [Figura 15-11](#): se utilizó `translate()` para mover el origen a la esquina inferior izquierda del lienzo, luego se utilizó `scale()` para voltear el eje `y` de manera que las coordenadas `y` aumenten a medida que subimos la página. Un sistema de coordenadas invertido como este es familiar desde la clase de álgebra y puede ser útil para trazar puntos de datos en los gráficos. Tenga en cuenta, sin embargo, que hace que el texto sea difícil de leer.

## COMPRENDER LAS TRANSFORMACIONES MATEMÁTICAMENTE

Me resulta más fácil entender las transformaciones geométricamente, pensando en `translate()`, `rotate()` y `scale()` como transformaciones de los ejes del sistema de coordenadas, como se ilustra en la [Figura 15-11](#). También es posible entender las transformaciones algebraicamente como ecuaciones que mapean las coordenadas de un punto  $(x,y)$  en el sistema de

coordenadas transformado a las coordenadas ( $x',y'$ ) del mismo punto en el sistema de coordenadas anterior.

La llamada al método `c.translate(dx,dy)` puede describirse con estas ecuaciones:

$x' = x + dx; // Una coordenada X de 0 en el nuevo sistema es dx en el antiguo$   $y' = y + dy;$

Las operaciones de escala tienen ecuaciones igualmente sencillas.

Una llamada `c.scale(sx,sy)` puede describirse así:

$x' = sx * x; y' = sy * y;$

Las rotaciones son más complicadas. La llamada `c.rotate(a)` se describe mediante estas ecuaciones trigonométricas:

$x' = x * \cos(a) - y * \sin(a); y' = y * \cos(a) + x * \sin(a);$

Observe que el orden de las transformaciones es importante.

Supongamos que empezamos con el sistema de coordenadas por defecto de un lienzo, luego lo trasladamos y luego lo escalamos.

Para mapear el punto  $(x,y)$  en el sistema de coordenadas actual de vuelta al punto  $(x'',y'')$  en el sistema de coordenadas por defecto, primero debemos aplicar las ecuaciones de escala para mapear el punto a un punto intermedio  $(x',y')$  en el sistema de coordenadas traducido pero no escalado, y luego usar las ecuaciones de translación para mapear desde este punto intermedio a  $(x'',y'')$ . El resultado es este:

$x'' = sx*x + dx; y'' = sy*y$   
+ dy;

Si, por el contrario, hubiéramos llamado a scale() antes de llamar a translate(), las ecuaciones resultantes serían diferentes:

$x'' = sx*(x + dx); y'' = sy*(y$   
+ dy);

La clave que hay que recordar cuando se piensa algebraicamente en secuencias de transformaciones es que hay que trabajar hacia atrás desde la última (más reciente) transformación hasta la primera. Sin embargo, cuando se piensa geométricamente en ejes transformados, se trabaja hacia adelante desde la primera transformación hasta la última.

Las transformaciones soportadas por el lienzo se conocen como *transformaciones afines*. Las transformaciones afines pueden modificar las distancias entre los puntos y los ángulos entre las líneas, pero las líneas paralelas siempre permanecen paralelas después de una transformación afín; no es posible, por ejemplo, especificar una distorsión de lente ojo de pez con una transformación afín. Una transformación afín arbitraria puede describirse mediante los seis parámetros de a a f de estas ecuaciones:

$x' = ax + cy + e$   $y' = bx +$   
 $dy + f$

Puede aplicar una transformación arbitraria al sistema de coordenadas actual pasando esos seis parámetros al método transform(). La Figura 15-11 ilustra dos tipos de transformaciones -

cortadas y rotaciones alrededor de un punto especificado- que puede implementar con el método transform() de esta manera:

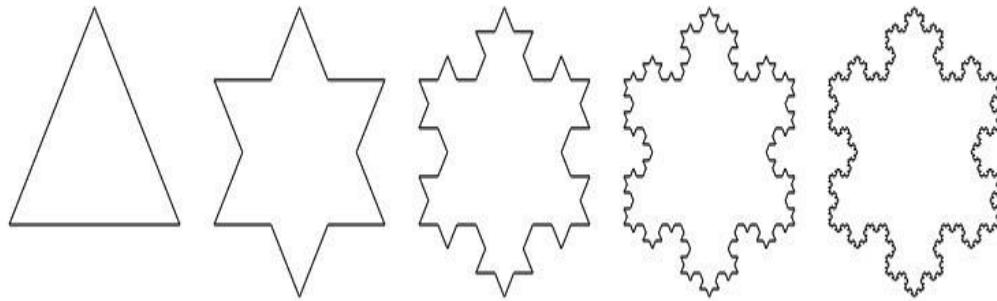
```
// Transformación de cizalla:  
//  $x' = x + kx*y$ ; //  $y' = ky*x + y$ ; function shear(c, kx, ky) { c. transform(1, ky, kx, 1, 0, 0); }  
  
// Gira theta radianes en sentido contrario a las agujas del reloj alrededor del punto  
(x,y)  
// Esto también puede lograrse con una secuencia de traslación, rotación y  
traslación function rotateAbout(c, theta, x, y) { let ct = Math. cos(theta); let st =  
Math. sin(theta);  
c. transformar(ct, -st, st, ct, -x*ct-y*st+x, x*st-y*ct+y);  
}
```

El método setTransform() toma los mismos argumentos que transform(), pero en lugar de transformar el sistema de coordenadas actual, ignora el sistema actual, transforma el sistema de coordenadas por defecto, y hace que el resultado sea el nuevo sistema de coordenadas actual. setTransform() es útil para restablecer temporalmente el lienzo a su sistema de coordenadas por defecto:

```
c. save(); // Guardar el sistema de coordenadas actual  
c. setTransform(1,0,0,1,0,0); // Volver al sistema de coordenadas por defecto //  
Realizar operaciones utilizando las coordenadas de píxeles por defecto de CSS  
c. restore(); // Restaurar el sistema de coordenadas guardado
```

## EJEMPLO DE TRANSFORMACIÓN

El Ejemplo 15-7 demuestra el poder de las transformaciones del sistema de coordenadas utilizando los métodos translate(), rotate() y scale() recursivamente para dibujar un fractal de copos de nieve Koch. La salida de este ejemplo aparece en la Figura 15-12, que muestra copos de nieve Koch con 0, 1, 2, 3 y 4 niveles de recursión.



*Figura 15-12. Copos de nieve Koch*

El código que produce estas figuras es elegante, pero su uso de transformaciones recursivas del sistema de coordenadas lo hace algo difícil de entender. Incluso si no sigue todos los matices, observe que el código incluye una sola invocación del método `lineTo()`. Cada segmento de línea en [la Figura 15-12](#) se dibuja así:

c. `lineTo(len, 0);`

El valor de la variable `len` no cambia durante la ejecución del programa, por lo que la posición, la orientación y la longitud de cada uno de los segmentos de la línea se determina mediante traslaciones, rotaciones y operaciones de escalado.

### *Ejemplo 15-7. Un copo de nieve de Koch con transformaciones*

---

```

let deg = Math.PI/180; // Para convertir los grados en radianes

// Dibuja un fractal de copo de nieve de nivel n Koch en el contexto del lienzo c, // con la
esquina inferior izquierda en (x,y) y la longitud del lado len. function snowflake(c, n, x, y,
len) {
    c. save(); // Guardar la transformación actual
    c. translate(x,y); // Trasladar el origen al punto de partida
    c. moveTo(0,0); // Comienza un nuevo subcamino en el nuevo origen leg(n); // Dibuja
el primer tramo del copo de nieve
    c. rotate(-120*deg); // Ahora gira 120 grados
}

```

```

pierna en sentido contrario a las agujas del reloj(n); // Dibuja la
segunda pierna
    c. rotate(-120*deg); // Rotar de nuevo leg(n); // Dibujar el
    tramo final
    c. closePath(); // Cerrar la sub-ruta
    c. restore(); // Y restaurar la transformación original

// Dibuja una sola pierna de un copo de nieve de nivel-n Koch.
// Esta función deja el punto actual al final del tramo que ha // dibujado y traslada el
sistema de coordenadas para que el punto actual sea (0,0). // Esto significa que puede
llamar fácilmente a rotate() después de dibujar un tramo.

función leg(n) {
    c. save(); // Guardar la transformación actual if (n === 0) { // Caso no
recursivo:
        c. lineTo(len, 0); // Sólo dibuja una línea horizontal }           // __ else { //
Caso recursivo: dibujar 4 subtemas como: \/
        c. scale(1/3,1/3); // Los subtramos son 1/3 del tamaño de este tramo leg(n-1); //
Recurrir al primer subtramo
        c. rotate(60*deg); // Gira 60 grados en el sentido de las agujas del reloj leg(n-1); //
Segunda sub-pata
        c. rotate(-120*deg); // Rotar 120 grados pata trasera(n-1); // Tercera pata
secundaria
        c. rotate(60*deg); // Gira de nuevo a nuestro rumbo original leg(n-1); // Sub-leg final
    }
    c. restore(); // Restaurar la transformación
    c. translate(len, 0); // Pero traduce para hacer el final del tramo (0,0) }
}

```



```
let c = document.querySelector("canvas").getContext("2d"); snowflake(c, 0, 25, 125, 125); // Un copo de nieve de nivel 0 es un triángulo snowflake(c, 1, 175, 125, 125); // Un copo de nieve de nivel 1 es una estrella de 6 lados snowflake(c, 2, 325, 125, 125); // etc. snowflake(c, 3, 475, 125, 125); snowflake(c, 4, 625, 125, 125); // ¡Un copo de nieve de nivel 4 parece un copo de nieve!  
c.stroke(); // Acaricia este camino tan complicado
```

## 15.8.6 Recorte

Después de definir un trazado, normalmente se llama a `stroke()` o `fill()` (o ambos). También puede llamar al método `clip()` para definir una región de recorte. Una vez definida una región de recorte, no se dibujará nada fuera de ella. [La Figura 15-13](#) muestra un dibujo complejo realizado con regiones de recorte. La franja vertical que corre por el centro y el texto a lo largo de la parte inferior de la figura fueron trazados sin región de recorte y luego rellenados después de definir la región de recorte triangular.

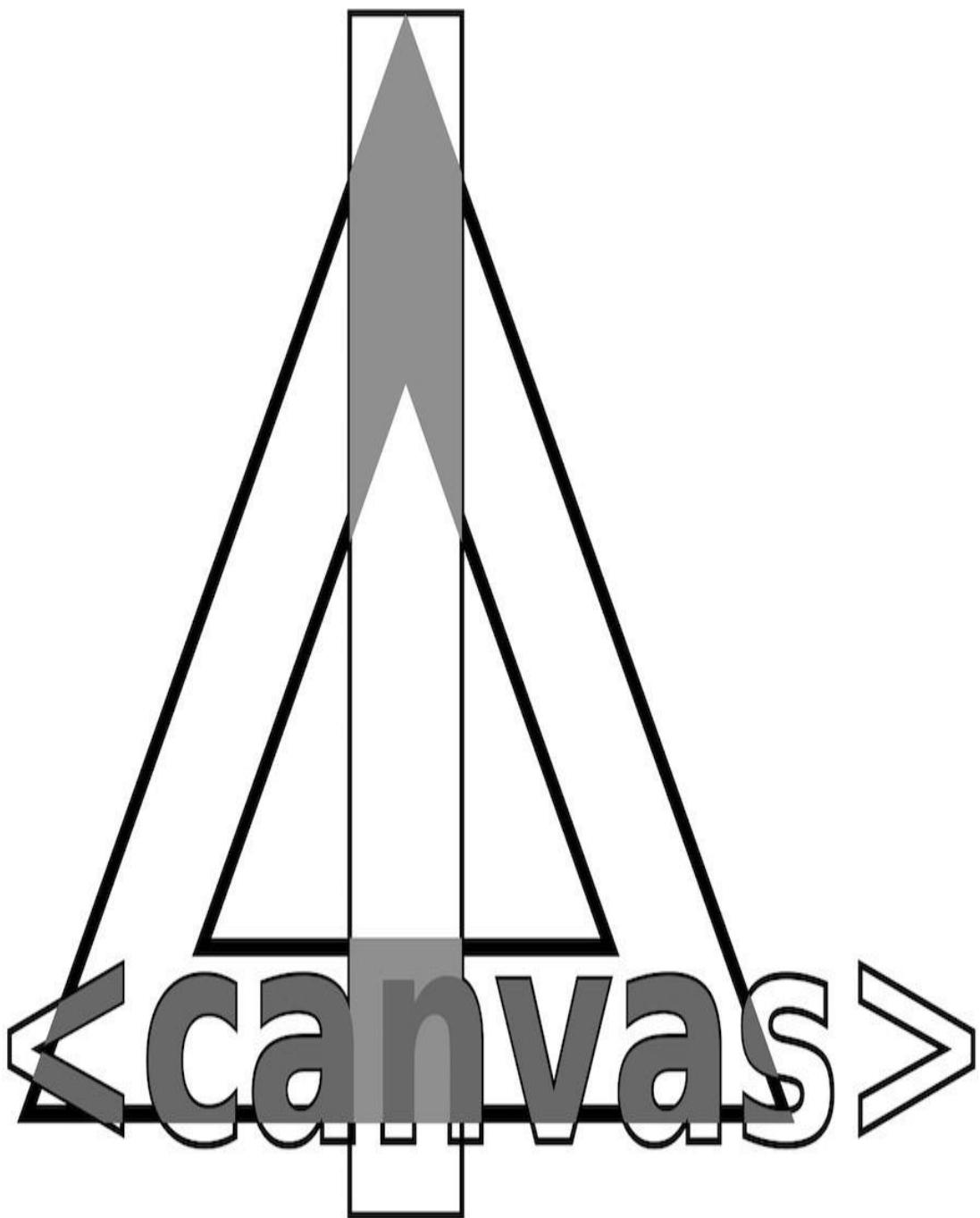


Figura 15-13. Trazos sin recortar y rellenos recortados

La Figura 15-13 fue generada usando el método `polygon()` del Ejemplo 15-5 y el siguiente código:

```
// Definir algunos atributos de dibujo
c. font = "bold 60pt sans-serif"; // Fuente grande
c. lineWidth = 2; // Líneas estrechas
c. strokeStyle = "#000"; // Líneas negras
```

```

// Contornear un rectángulo y un texto
c. strokeRect(175, 25, 50, 325); // Una franja vertical en el centro
c. strokeText("<lienzo>", 15, 330); // Observe strokeText() en lugar de fillText()

// Definir un camino complejo con un interior que está fuera. polygon(c,3,200,225,200);
// Triángulo grande polygon(c,3,200,225,100,0,true); // Triángulo inverso más pequeño
dentro

// Haz que ese camino sea la región de recorte.
c. clip();

// Trazar la ruta con una línea de 5 píxeles, completamente dentro de la región de
recorte.
c. lineWidth = 10; // La mitad de esta línea de 10 píxeles se recortará
c. stroke();

// Rellenar las partes del rectángulo y del texto que están dentro de la región de recorte
c. fillStyle = "#aaa"; // Gris claro
c. fillRect(175, 25, 50, 325); // Rellenar la franja vertical
c. fillStyle = "#888"; // Gris más oscuro
c. fillText("<lienzo>", 15, 330); // Rellenar el texto

```

Es importante tener en cuenta que cuando se llama a `clip()`, la ruta actual se recorta a su vez a la región de recorte actual, y luego esa ruta recortada se convierte en la nueva región de recorte. Esto significa que el método `clip()` puede reducir la región de recorte pero nunca puede ampliarla. No hay ningún método para restablecer la región de recorte, así que antes de llamar a `clip()`, debería llamar a `save()` para poder restaurar() más tarde la región no recortada.

### 15.8.7 Manipulación de píxeles

El método `getImageData()` devuelve un objeto `ImageData` que representa los píxeles en bruto (como componentes R, G, B y A) de una región rectangular de su lienzo. Puede crear objetos `ImageData` vacíos con `createImageData()`. Los píxeles en un objeto `ImageData` son escribibles, por lo que puede establecerlos de

cualquier manera que deseé, y luego copiar esos píxeles de nuevo en el lienzo con `putImageData()`.

Estos métodos de manipulación de píxeles proporcionan un acceso de muy bajo nivel al lienzo. El rectángulo que se pasa a `getImageData()` está en el sistema de coordenadas por defecto: sus dimensiones se miden en píxeles CSS, y no se ve afectado por la transformación actual. Cuando se llama a `putImageData()`, la posición que se especifica también se mide en el sistema de coordenadas por defecto. Además, `putImageData()` ignora todos los atributos gráficos. No realiza ninguna composición, no multiplica los píxeles por `globalAlpha` y no dibuja sombras.

Los métodos de manipulación de píxeles son útiles para implementar el procesamiento de imágenes. [El Ejemplo 15-8](#) muestra cómo crear un simple efecto de desenfoque de movimiento o "smear" como el que se muestra en [la Figura 15-14](#).



*Figura 15-14. Un efecto de desenfoque de movimiento creado por el procesamiento de imágenes*

El siguiente código demuestra `getImageData()` y `putImageData()` y muestra cómo iterar y modificar los valores de los píxeles en un objeto `ImageData`.

*Ejemplo 15-8. Desenfoque de movimiento con `ImageData`*

```

// Difumina los píxeles del rectángulo hacia la derecha, produciendo una // especie de
desenfoque de movimiento como si los objetos se movieran de derecha a izquierda. // n
debe ser 2 o mayor. Los valores más grandes producen manchas más grandes.
// El rectángulo se especifica en el sistema de coordenadas por defecto. function smear(c,
n, x, y, w, h) {
    // Obtener el objeto ImageData que representa el rectángulo de píxeles a manchar let
pixels = c.getImageData(x, y, w, h);

    // Este frotado se hace en el lugar y requiere sólo el ImageData de origen.
    // Algunos algoritmos de procesamiento de imágenes requieren un
ImageData a
    // almacenar los valores de los píxeles transformados. Si necesitáramos un buffer de
salida, podríamos // crear un nuevo ImageData con las mismas dimensiones de la
siguiente manera: // let output_pixels = c.createImageData(pixels);

    // Obtener las dimensiones de la cuadrícula de píxeles en el
Objeto ImageData deje que width = pixels. width, height = pixels. height;

    // Esta es la matriz de bytes que contiene los datos de los píxeles en bruto, de
izquierda a derecha y // de arriba a abajo. Cada píxel ocupa 4 bytes consecutivos en
orden R,G,B,A. let data = pixels. data;

    // Cada píxel después del primero en cada fila se emborrona sustituyéndolo por
// 1/nº de su propio valor más m/nº del valor del pixel anterior let m = n-1;

    for(let row = 0; row < height; row++) { // Para cada fila let i = row*width*4 + 4; // El
desplazamiento del segundo píxel de la fila for(let col = 1; col < width; col++, i += 4) { //
Para cada columna data[i] = (data[i] + data[i-4]*m)/n; // Componente de píxel rojo
data[i+1] = (data[i+1] + data[i-3]*m)/n; // Verde data[i+2] = (data[i+2] + data[i-2]*m)/n;
// Azul data[i+3] = (data[i+3] + data[i-1]*m)/n; // Componente alfa }
}

    // Ahora copie los datos de la imagen manchada de nuevo a la misma posición en el
lienzo
c. putImageData(pixels, x, y); }

```

## 15.9 APIs de audio

Las etiquetas HTML <audio> y <video> permiten incluir fácilmente sonido y vídeos en las páginas web. Se trata de elementos complejos con importantes APIs e interfaces de usuario no triviales. Puedes controlar la reproducción de medios con los métodos play() y pause(). Puedes establecer las propiedades volume y playbackRate para controlar el volumen del audio y la velocidad de reproducción. Y puedes saltar a un momento concreto dentro del medio estableciendo la propiedad currentTime.

Sin embargo, no trataremos aquí las etiquetas <audio> y <video> con más detalle. En las siguientes subsecciones se muestran dos formas de añadir efectos de sonido con guión a las páginas web.

### 15.9.1 El constructor Audio()

No es necesario incluir una etiqueta <audio> en el documento HTML para incluir efectos de sonido en las páginas web. Puede crear dinámicamente elementos <audio> con el método normal DOM document.createElement() o, como atajo, puede utilizar simplemente el constructor Audio(). No es necesario añadir el elemento creado a su documento para reproducirlo. Puedes simplemente llamar a su método play():

```
// Cargar el efecto de sonido por adelantado para que esté listo para su uso let  
soundeffect = new Audio("soundeffect.mp3");
```

```
// Reproducir el efecto de sonido cada vez que el usuario haga clic en el botón del  
ratón document.addEventListener("click", () => { soundeffect.cloneNode().play();  
// Cargar y reproducir el sonido });
```

Observe el uso de `cloneNode()` aquí. Si el usuario hace clic en el ratón rápidamente, queremos ser capaces de tener múltiples copias superpuestas del efecto de sonido que se reproduce al mismo tiempo. Para ello, necesitamos múltiples elementos de Audio. Dado que los elementos de audio no se añaden al documento, serán recogidos por la basura cuando terminen de reproducirse.

### 15.9.2 La API WebAudio

Además de la reproducción de sonidos grabados con elementos de audio, los navegadores web también permiten la generación y reproducción de sonidos sintetizados con la API WebAudio. Utilizar la API de WebAudio es como conectar un sintetizador electrónico de estilo antiguo con cables de conexión. Con WebAudio, se crea un conjunto de objetos `AudioNode`, que representan fuentes, transformaciones o destinos de formas de onda, y luego se conectan estos nodos en una red para producir sonidos. La API no es particularmente compleja, pero una explicación completa requiere una comprensión de la música electrónica y de los conceptos de procesamiento de señales que están más allá del alcance de este libro.

El siguiente código utiliza la API de WebAudio para sintetizar un acorde corto que se desvanece en aproximadamente un segundo. Este ejemplo demuestra los fundamentos de la API de WebAudio. Si le resulta interesante, puede encontrar mucho más sobre esta API en Internet:

```
// Comienza creando un objeto audioContext. Safari todavía requiere
// que usemos webkitAudioContext en lugar de AudioContext.
let audioContext = new (this. AudioContext | |this. webkitAudioContext)();

// Definir el sonido base como una combinación de tres ondas sinusoidales puras let
notes = [ 293.7, 370.0, 440.0 ]; // Acorde de re mayor: Re, Fa# y La

// Crear nodos osciladores para cada una de las notas que queremos reproducir let
oscillators = notes. map(note => { let o = audioContext. createOscillator();
o. frecuencia. valor = nota; return o; });

// Da forma al sonido controlando su volumen en el tiempo.
// Empezando en el momento 0, sube rápidamente el volumen.
// A continuación, a partir del tiempo 0,1, baja lentamente hasta 0. let
volumeControl = audioContext. createGain(); volumeControl. gain.
setTargetAtTime(1, 0, 0, 0,02); volumeControl. gain. setTargetAtTime(0, 0,1,
0,2);

// Vamos a enviar el sonido al destino por defecto:
// los altavoces del usuario let speakers = audioContext.
destination;
```

```

// Conecta cada una de las notas de origen a los osciladores de control de volumen.
forEach(o => o.connect(volumeControl));

// Y conecta la salida del control de volumen a los altavoces. volumeControl.
connect(speakers);

// Ahora empieza a reproducir los sonidos y deja que se ejecuten durante 1,25 segundos.
let startTime = audioContext.currentTime; let stopTime =
startTime + 1.25; oscillators.forEach(o => {
  o.start(startTime);
  o.stop(stopTime); });

// Si queremos crear una secuencia de sonidos podemos usar manejadores de eventos
oscillators[0].addEventListener("ended", () => {
  // Este controlador de eventos se invoca cuando la nota deja de sonar
});

```

## 15.10 Localización, navegación e historia

La propiedad location de los objetos Window y Document hace referencia al objeto Location, que representa la URL actual del documento mostrado en la ventana, y que también proporciona una API para cargar nuevos documentos en la ventana.

El objeto Location es muy parecido a un objeto URL ([§11.9](#)), y puede utilizar propiedades como protocolo, nombre de host, puerto y ruta para acceder a las distintas partes de la URL del documento actual. La propiedad href devuelve la URL completa como una cadena, al igual que el método `toString()`.

Las propiedades hash y de búsqueda del objeto Location son interesantes. La propiedad hash devuelve la parte del "identificador de fragmentos" de la URL, si existe: una marca de

hash (#) seguida de un ID de elemento. La propiedad de búsqueda es similar. Devuelve la parte de la URL que comienza con un signo de interrogación: a menudo algún tipo de cadena de consulta. En general, esta porción de una URL se utiliza para parametrizar la URL y proporciona una forma de incrustar argumentos en ella. Aunque estos argumentos suelen estar destinados a los scripts que se ejecutan en un servidor, no hay ninguna razón por la que no puedan utilizarse también en páginas con JavaScript.

Los objetos URL tienen una propiedad searchParams que es una representación analizada de la propiedad de búsqueda. El objeto Location no tiene una propiedad searchParams, pero si quieres analizar window.location.search, puedes simplemente crear un objeto URL a partir del objeto Location y luego utilizar los searchParams de la URL:

```
let url = new URL(window.location); let query = url.searchParams.  
get("q");  
let numResults = parseInt(url.searchParams.get("n") || "10");
```

Además del objeto Location al que se puede referir como window.location o document.location, y el constructor URL() que usamos antes, los navegadores también definen una propiedad document.URL. Sorprendentemente, el valor de esta propiedad no es un objeto URL, sino simplemente una cadena. La cadena contiene la URL del documento actual.

### 15.10.1 Carga de nuevos documentos

Si asignas una cadena a window.location o a document.location, esa cadena se interpreta como una URL y el navegador la carga, sustituyendo el documento actual por uno nuevo:

```
window. location = "http://www.oreilly.com"; // iVaya a comprar algunos libros!
```

También puede asignar URLs relativas a la ubicación. Se resuelven de forma relativa a la URL actual:

```
document. location = "page2.html"; // Cargar la siguiente página
```

Un identificador de fragmento desnudo es un tipo especial de URL relativa que no hace que el navegador cargue un nuevo documento, sino que simplemente se desplaza para que el elemento del documento con id o nombre que coincide con el fragmento sea visible en la parte superior de la ventana del navegador. Como caso especial, el identificador de fragmento #top hace que el navegador salte al principio del documento (suponiendo que ningún elemento tenga un atributo id="top"):

```
location = "#top"; // Saltar a la parte superior del documento
```

Las propiedades individuales del objeto Location son escribibles, y al establecerlas se cambia la URL de la ubicación y también se hace que el navegador cargue un nuevo documento (o, en el caso de la propiedad hash, que navegue dentro del documento actual):

```
document. location. path = "pages/3.html"; // Cargar una nueva página  
document. location. hash = "TOC"; // Desplazarse hasta el índice location. search  
= "?page=" + (page+1); // Recargar con la nueva cadena de consulta
```

También puede cargar una nueva página pasando una nueva cadena al método assign() del objeto Location. Sin embargo, esto

es lo mismo que asignar la cadena a la propiedad location, por lo que no es especialmente interesante.

Por otro lado, el método replace() del objeto Location es bastante útil. Cuando se pasa una cadena a replace(), se interpreta como una URL y hace que el navegador cargue una nueva página, al igual que hace assign(). La diferencia es que replace() reemplaza el documento actual en el historial del navegador. Si un script en el documento A establece la propiedad location o llama a assign() para cargar el documento B y luego el usuario hace clic en el botón Atrás, el navegador volverá al documento A. Si en cambio se utiliza replace(), entonces el documento A se borra del historial del navegador, y cuando el usuario hace clic en el botón Atrás, el navegador vuelve al documento que se mostraba antes del documento A.

Cuando un script carga incondicionalmente un nuevo documento, el método replace() es una mejor opción que assign(). De lo contrario, el botón Atrás llevaría al navegador de vuelta al documento original, y el mismo script volvería a cargar el nuevo documento. Suponga que tiene una versión mejorada con JavaScript de su página y una versión estática que no utiliza JavaScript. Si determina que el navegador del usuario no soporta las APIs de la plataforma web que desea utilizar, podría utilizar location.replace() para cargar la versión estática:

```
// Si el navegador no soporta las APIs de JavaScript que necesitamos, // redirige a  
una página estática que no utiliza JavaScript.  
si (! isBrowserSupported())  
location.replace("staticpage.html");
```

Observe que la URL pasada a replace() es relativa. Las URLs relativas se interpretan de forma relativa a la página en la que aparecen, igual que si se utilizaran en un hipervínculo.

Además de los métodos assign() y replace(), el objeto Location también define reload(), que simplemente hace que el navegador recargue el documento.

### 15.10.2 Historial de navegación

La propiedad history del objeto Window hace referencia al objeto History de la ventana. El objeto History modela el historial de navegación de una ventana como una lista de documentos y estados de documentos. La propiedad length del objeto History especifica el número de elementos de la lista del historial de navegación, pero por razones de seguridad, los scripts no pueden acceder a las URLs almacenadas. (Si pudieran, cualquier script podría husmear en el historial de navegación).

El objeto History tiene métodos back() y forward() que se comportan como los botones Back y Forward del navegador: hacen que el navegador retroceda o avance un paso en su historial de navegación. Un tercer método, go(), toma un argumento entero y puede saltar cualquier número de páginas hacia adelante (para argumentos positivos) o hacia atrás (para argumentos negativos) en la lista del historial:

```
history.go(-2); // Retrocede 2, como si hicieras dos veces el botón Atrás history.  
go(0); // Otra forma de recargar la página actual
```

Si una ventana contiene ventanas hijas (como elementos <iframe>), los históricos de navegación de las ventanas hijas se intercalan cronológicamente con el historial de la ventana

principal. Esto significa que llamar a `history.back()` (por ejemplo) en la ventana principal puede hacer que una de las ventanas hijas retroceda a un documento mostrado anteriormente, pero deja la ventana principal en su estado actual.

El objeto History descrito aquí se remonta a los primeros días de la web, cuando los documentos eran pasivos y todo el cálculo se realizaba en el servidor. Hoy en día, las aplicaciones web a menudo generan o cargan contenido dinámicamente y muestran nuevos estados de la aplicación sin cargar realmente nuevos documentos. Este tipo de aplicaciones deben realizar su propia gestión del historial si quieren que el usuario pueda utilizar los botones Atrás y Adelante (o los gestos equivalentes) para navegar de un estado de la aplicación a otro de forma intuitiva. Hay dos maneras de conseguirlo, que se describen en las dos secciones siguientes.

### **15.10.3 Gestión del historial con eventos de intercambio de hash**

Una de las técnicas de gestión del historial es `location.hash` y el evento "hashchange". Estos son los datos clave que debes conocer para entender esta técnica:

- La propiedad `location.hash` establece el identificador del fragmento de la URL y se utiliza tradicionalmente para especificar el ID de una sección del documento a la que desplazarse. Pero `location.hash` no tiene por qué ser el ID de un elemento: puedes ponerle cualquier cadena. Mientras ningún elemento tenga esa cadena como ID, el navegador no se desplazará cuando se establezca la propiedad `hash` como

esto.

- Al establecer la propiedad location.hash se actualiza la URL que aparece en la barra de direcciones y, lo que es muy importante, se añade una entrada al historial del navegador.
- Cada vez que el identificador del fragmento del documento cambia, el navegador dispara un evento "hashchange" en el objeto Window. Si se establece location.hash explícitamente, se dispara un evento "hashchange". Y, como hemos mencionado, este cambio en el objeto Location crea una nueva entrada en el historial de navegación del navegador. Así que si el usuario ahora hace clic en el botón Atrás, el navegador volverá a su URL anterior antes de establecer location.hash. Pero esto significa que el identificador del fragmento ha cambiado de nuevo, por lo que en este caso se dispara otro evento "hashchange". Esto significa que mientras puedas crear un identificador de fragmento único para cada estado posible de tu aplicación, los eventos "hashchange" te notificarán si el usuario se mueve hacia atrás y hacia delante a través de su historial de navegación.

Para utilizar este mecanismo de gestión del historial, tendrás que ser capaz de codificar la información de estado necesaria para representar una "página" de tu aplicación en una cadena de texto relativamente corta que sea adecuada para su uso como identificador de fragmento. Y tendrás que escribir una función para convertir el estado de la página en una cadena y otra función para analizar la cadena y recrear el estado de la página que representa.

Una vez escritas esas funciones, el resto es fácil. Defina una función window.onhashchange (o registre un oyente de "hashchange" con addEventListener()) que lea

`location.hash`, convierte esa cadena en una representación del estado de su aplicación, y luego realiza las acciones necesarias para mostrar ese nuevo estado de la aplicación.

Cuando el usuario interactúa con tu aplicación (por ejemplo, haciendo clic en un enlace) de manera que la aplicación entre en un nuevo estado, no renderices el nuevo estado directamente. En su lugar, codifica el nuevo estado deseado como una cadena y establece `location.hash` a esa cadena. Esto desencadenará un evento "hashchange", y su controlador para ese evento mostrará el nuevo estado. El uso de esta técnica indirecta asegura que el nuevo estado se inserte en el historial de navegación para que los botones de Atrás y Adelante sigan funcionando.

#### **15.10.4 Gestión del historial con `pushState()`**

La segunda técnica de gestión del historial es algo más compleja, pero no es tan complicada como el evento "hashchange". Esta técnica más robusta de gestión del historial se basa en el `history.pushState()` y el evento "popstate". Cuando una aplicación web entra en un nuevo estado, llama a `history.pushState()` para añadir un objeto que representa el estado al historial del navegador. Si el usuario hace clic en el botón Atrás, el navegador lanza un evento "popstate" con una copia de ese objeto de estado guardado, y la aplicación utiliza ese objeto para recrear su estado anterior. Además del objeto de estado guardado, las aplicaciones también pueden guardar una URL con cada estado, lo cual es importante si quieras que los usuarios puedan marcar y compartir enlaces a los estados internos de la aplicación.

El primer argumento de `pushState()` es un objeto que contiene toda la información de estado necesaria para restaurar el estado actual del documento. Este objeto se guarda utilizando el algoritmo de *clonación estructurada* de HTML, que es más versátil que `JSON.stringify()` y puede soportar objetos Map, Set y Date, así como arrays tipificados y ArrayBuffers.

### EL ALGORITMO DE CLONACIÓN ESTRUCTURADA

El método `history.pushState()` no utiliza `JSON.stringify()` (§11.6\_\_\_\_\_.) para serializar los datos de estado. En su lugar, éste (y otras APIs del navegador que conoceremos más adelante) utiliza una técnica de serialización más robusta conocida como algoritmo de clonación estructurada, definida por el estándar HTML.

El algoritmo de clonación estructurada puede serializar cualquier cosa que `JSON.stringify()` pueda, pero además, permite la serialización de la mayoría de los otros tipos de JavaScript, incluyendo Map, Set, Date, RegExp, y arrays tipados, y puede manejar estructuras de datos que incluyen referencias circulares. Sin embargo, el algoritmo de clonación estructurada *no puede* serializar funciones o clases. Al clonar objetos no copia el prototipo del objeto, los getters y setters, ni las propiedades no numerables. Aunque el algoritmo de clonación estructurada puede clonar la mayoría de los tipos incorporados de JavaScript, no puede copiar los tipos definidos por el entorno anfitrión, como los objetos `document` Element.

Esto significa que el objeto de estado que se pasa a `history.pushState()` no tiene por qué limitarse a los objetos, matrices y valores primitivos que admite `JSON.stringify()`. Tenga en cuenta, sin embargo, que si pasa una instancia de una clase que haya definido, esa instancia se serializará como un objeto JavaScript ordinario y perderá su prototipo.

El segundo argumento estaba destinado a ser una cadena de título para el estado, pero la mayoría de los navegadores no lo soportan, y se debe pasar simplemente una cadena vacía. El tercer argumento es una URL opcional que se mostrará en la barra de localización inmediatamente y también si el usuario vuelve a este estado a través de los botones Atrás y Adelante. Las URLs relativas se resuelven contra la ubicación actual del documento. Asociar una URL a cada estado permite al usuario marcar estados internos de su aplicación. Recuerda, sin embargo, que si el usuario guarda un marcador y lo visita un día después,

no recibirás un evento "popstate" sobre esa visita: tendrás que restaurar el estado de tu aplicación analizando la URL.

Además del método `pushState()`, el objeto `History` también define `replaceState()`, que toma los mismos argumentos pero reemplaza el estado actual del historial en lugar de añadir un nuevo estado al historial de navegación. Cuando una aplicación que utiliza `pushState()` se carga por primera vez, a menudo es

una buena idea llamar a `replaceState()` para definir un objeto de estado para este estado inicial de la aplicación.

Cuando el usuario navega a los estados guardados del historial usando los botones Atrás o Adelante, el navegador dispara un evento "popstate" en el objeto Window. El objeto de evento asociado con el evento tiene una propiedad llamada `state`, que contiene una copia (otro clon estructurado) del objeto `state` que pasaste a `pushState()`.

El ejemplo 15-9 es una sencilla aplicación web -el juego de adivinar números que se muestra en la Figura 15-15- que utiliza `pushState()` para guardar su historial, permitiendo al usuario "volver atrás" para revisar o rehacer sus conjeturas.

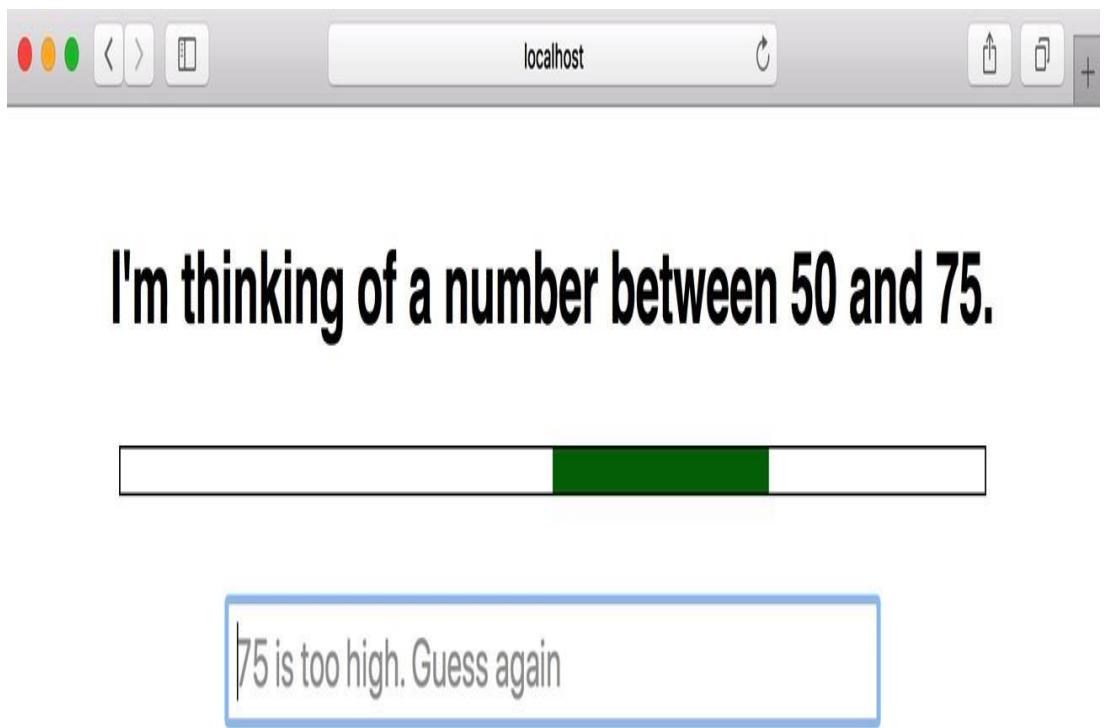


Figura 15-15. Un juego de adivinación de números

Ejemplo 15-9. Gestión del historial con `pushState()`

---

```
<html><head><title>Estoy pensando en un número...</title>
<style>
```

```

body { height: 250px; display: flex; flex-direction: column; align-items: center; justify-content: space-evenly; }
#heading { font: bold 36px sans-serif; margin: 0; }
#container { borde: negro sólido 1px; altura: 1em; anchura: 80%; }
#range { color de fondo: verde; margin-left: 0%; height: 1em; width: 100%; }
#input { display: block; font-size: 24px; width: 60%; padding: 5px; }
#playagain { font-size: 24px; padding: 10px; border-radius: 5px; }

</style>
</head>
<cuerpo>
<h1 id="heading"> Estoy pensando en un número... </h1>
Una representación visual de los números que no han sido descartados -->
<div id="contenedor"><div id="rango"></div></div> <!-- Donde el usuario introduce su suposición -->
<input id="input" type="text">
<br /> Un botón que se recarga sin cadena de búsqueda. Oculto hasta que el juego termine. -->
<button id="playagain" hidden onclick="location.search='';"> Play Again</button>
<script>
/**
 * Una instancia de esta clase GameState representa el estado interno de nuestro juego de adivinar números. La clase define métodos estáticos de fábrica para inicializar el estado del juego a partir de diferentes fuentes, un método para actualizar el estado basado en una nueva suposición, y un método para modificar el documento basado en el estado actual.
 */
class GameState { // Esta es una función de fábrica para crear un nuevo juego
static newGame() { let s = new GameState();
    s.secret = s.randomInt(0, 100); // Un número entero: 0 < n < 100
    s.low = 0; // Las conjeturas deben ser mayores que esto
}

```

```
s. alto = 100; // Las conjeturas deben ser menores que esto  
s. numGuesses = 0; // Cuántas adivinanzas se han hecho  
s. guess = null; // Cuál fue la última suposición return s; }
```

*Cuando guardamos el estado del juego con history.pushState(), es sólo // un objeto JavaScript que se guarda, no una instancia de GameState.*

*// Así que esta función de fábrica recrea un objeto GameState basado en el // objeto simple que obtenemos de un evento popstate. static fromStateObject(stateObject) { let s = new GameState(); for(let key of Object. keys(stateObject)) { s[key] = stateObject[key]; } return s; }*

*// Para habilitar los marcadores, necesitamos ser capaces de codificar el // estado de cualquier juego como una URL. Esto es fácil de hacer con*

*URLSearchParams. toURL() { let url = new URL(window. location); url. searchParams. set("l", this. low); url. searchParams. set("h", this. high); url. searchParams. set("n", this. numGuesses); url. searchParams. set("g", this. guess);*

*// Tenga en cuenta que no podemos codificar el número secreto en la url o // revelará el secreto. Si el usuario marca la página con // estos parámetros y luego vuelve a él, simplemente escogeremos un // nuevo número aleatorio entre el bajo y el alto. return url. href;*

```
}
```

```

// Esta es una función de fábrica que crea un nuevo objeto GameState y
// lo inicializa a partir de la URL especificada. Si la URL no contiene los parámetros //
esperados o si están mal formados simplemente devuelve null.

static fromURL(url) { let s = new GameState(); let params = new
URL(url).searchParams;
  s. low = parseInt(params.get("l"));
  s. high = parseInt(params.get("h"));
  s. numGuesses = parseInt(params.get("n"));
  s. guess = parseInt(params.get("g"));

// Si a la URL le falta alguno de los parámetros que necesitamos o si // no se analizan
como enteros, entonces devuelve null; if (isNaN(s. low) | isNaN(s. high) || isNaN(s.
numGuesses) | isNaN(s. guess)) { return null; }

// Elige un nuevo número secreto en el rango correcto cada vez que //
restauramos un juego desde una URL.
  s. secret = s. randomInt(s. low, s. high); return s; }

// Devuelve un número entero n, min < n < max randomInt(min, max) {
  return min + Math. ceil(Math. random() * (max - min - 1));
}

// Modificar el documento para mostrar el estado actual del juego.
render() {
  let heading = document. querySelector("#heading"); // El <h1> de la parte superior
  let range = document. querySelector("#range"); //
  Visualizar el rango de adivinación
}

```

```
let input = document.querySelector("#input"); // Adivinar campo de entrada
let playagain = document.querySelector("#playagain");

// Actualizar el encabezamiento del documento y el título. textContent = document.
title =
    `Estoy pensando en un número entre ${this.low} y
${este.alto}.`;

// Actualizar el rango visual de los números range.style.marginLeft = `${this.low}%`;
range.style.width = `${(this.highthis.low)}%`;

// Asegúrese de que el campo de entrada está vacío y enfocado. input.value =
""; input.focus();

// Mostrar la información basada en la última suposición del usuario. El marcador
de entrada // se mostrará porque hemos dejado el campo de entrada vacío.
if (this.guess === null) {
    input.placeholder = "Escriba su respuesta y pulse Enter"; } else if (this.guess < this.
secret) { input.placeholder = `${this.guess} es demasiado bajo. Adivina de nuevo`; } else if
(this.guess > this.secret) { input.placeholder = `${this.guess} is too high. Adivina de
nuevo`; } else { input.placeholder = document.title = `${esta.adivinación} es correcta!`;
heading.textContent = `¡Ganas en ${this.numGuesses} adivinaciones!`; playagain.hidden
= false; }

// Actualiza el estado del juego basado en lo que el usuario adivinó. // Devuelve
true si el estado fue actualizado, y false en caso contrario.
```

```
updateForGuess(guess) { // Si es un número y está en el rango correcto if ((guess > this. low) && (guess < this. high)) { // Actualiza el objeto de estado basándose en esta guess if (guess < this. secret) this. low = guess; else if (guess > this. secret) this. high = guess; this. guess = guess; this. numGuesses++; return true; }

else { // Una suposición no válida: notifica al usuario pero no actualiza el
estado alert(`Por favor, introduzca un número mayor que ${this. low} y menor que
${this. high}`); return false; }

}

// Con la clase GameState definida, hacer que el juego funcione es sólo cuestión de
// de inicializar, actualizar, guardar y renderizar el objeto de estado en // los
momentos adecuados.

// Cuando se carga por primera vez, intentamos obtener el estado del juego desde la URL
// y si eso falla, comenzamos una nueva partida. Así que si el usuario marca un
// juego ese juego puede ser restaurado desde la URL. Pero si cargamos una página sin //
parámetros de consulta sólo obtendremos un nuevo juego.

let gamestate = GameState. fromURL(window. location) || GameState. newGame();

// Guarda este estado inicial del juego en el historial del navegador, pero utiliza //
replaceState en lugar de pushState() para este historial inicial de la página.
replaceState(gamestate, "", gamestate. toURL());

// Mostrar este estado inicial gamestate. render();
```

```

// Cuando el usuario adivina, actualiza el estado del juego en base a su adivinación
// entonces guarda el nuevo estado en el historial del navegador y renderiza el nuevo estado
document. querySelector("#input"). onchange = (event) => { if (gamestate.
updateForGuess(parseInt(event. target. value))) { history. pushState(gamestate, "", gamestate. toURL()); }
gamestate. render(); };

// Si el usuario retrocede o avanza en el historial, obtendremos un evento popstate
// en el objeto ventana con una copia del objeto estado que guardamos con //
pushState. Cuando eso ocurra, renderiza el nuevo estado.
window. onpopstate = (event) => {
  gamestate = GameState. fromStateObject(event. state); // Restaurar el estado
  gamestate. render(); // y mostrarlo };
</script>
</body></html>

```

## 15.11 Creación de redes

Cada vez que se carga una página web, el navegador realiza peticiones de red -utilizando los protocolos HTTP y HTTPS- para un archivo HTML, así como para las imágenes, fuentes, scripts y hojas de estilo de las que depende el archivo. Pero además de poder realizar peticiones de red en respuesta a las acciones del usuario, los navegadores web también exponen APIs de JavaScript para la creación de redes.

Esta sección cubre tres APIs de red:

- El método `fetch()` define una API basada en promesas para realizar peticiones HTTP y HTTPS. La API `fetch()` hace que las solicitudes GET básicas sean sencillas, pero tiene un amplio

conjunto de características que también soporta casi cualquier caso de uso HTTP posible.

- La API de eventos enviados por el servidor (o SSE) es una cómoda interfaz basada en eventos para las técnicas de "sondeo largo" de HTTP, en las que el servidor web mantiene la conexión de red abierta para poder enviar datos al cliente cuando lo deseé.
- WebSockets es un protocolo de red que no es HTTP, pero está diseñado para interoperar con HTTP. Define una API asíncrona de paso de mensajes en la que los clientes y los servidores pueden enviar y recibir mensajes entre sí de forma similar a los sockets de red TCP.

### 15.11.1 fetch()

Para las peticiones HTTP básicas, el uso de `fetch()` es un proceso de tres pasos:

1. Llama a `fetch()`, pasando la URL cuyo contenido quieras recuperar.
2. Obtener el objeto de respuesta que es devuelto asincrónicamente por el paso 1 cuando la respuesta HTTP comienza a llegar y llamar a un método de este objeto de respuesta para pedir el cuerpo de la respuesta.
3. Obtenga el objeto cuerpo devuelto de forma asíncrona por el paso 2 y procéselo como quiera.

La API `fetch()` está completamente basada en `Promise`, y hay dos pasos asíncronos aquí, por lo que normalmente se esperan dos llamadas `then()` o dos expresiones `await` cuando se utiliza `fetch()`. (Y si has olvidado lo que son, puede que quieras releer [el capítulo 13](#) antes de continuar con esta sección).

Esto es lo que parece una petición `fetch()` si estás usando `then()` y esperas que la respuesta del servidor a tu petición tenga formato JSON:

```
fetch("/api/users/current") // Realiza una petición GET HTTP (o HTTPS)
  .then(respuesta => respuesta.json()) // Analiza su cuerpo como un
  Objeto JSON .then(currentUser => { // Luego procesa ese objeto analizado
    displayUserInfo(currentUser);
 });
```

A continuación se muestra una petición similar realizada con las palabras clave `async` y `await` a una API que devuelve una cadena simple en lugar de un objeto JSON:

```
async function isServiceReady() { let response = await
fetch("/api/service/status"); let body = await response.text(); return body
 === "ready"; }
```

Si entiendes estos dos ejemplos de código, entonces sabes el 80% de lo que necesitas saber para usar la API `fetch()`. Las subsecciones que siguen demostrarán cómo hacer peticiones y recibir respuestas que son algo más complicadas que las

#### ADIÓS XMLHTTPREQUEST

La API `fetch()` sustituye a la barroca y engañosamente llamada API `XMLHttpRequest` (que no tiene nada que ver con XML). Es posible que todavía veas XHR (como se abrevia a menudo) en el código existente, pero no hay ninguna razón hoy en día para usarlo en el código nuevo, y no está documentado en este capítulo. Sin embargo, hay un ejemplo de `XMLHttpRequest` en este libro, y puedes consultar §13.1.3 siquieres ver un

-----  
ejemplo de red JavaScript a la antigua usanza.

mostradas aquí.

## CÓDIGOS DE ESTADO HTTP, CABECERAS DE RESPUESTA Y ERRORES DE RED

El proceso de tres pasos de fetch() mostrado en §15.11.1 elude todo el código de tratamiento de errores. Aquí hay una versión más realista:

```
fetch("/api/users/current") // Realiza una petición GET HTTP (o HTTPS). . .
then(response => { // Cuando obtenemos una respuesta, primero la comprobamos si
  (response.ok && // para un código de éxito y el tipo esperado.
   response.headers.get("Content-Type") === "application/json") { return
  response.json(); // Devuelve una Promise para el cuerpo. } } else { throw new
  Error(` // O lanza un error. Estado de la respuesta no esperado
  ${response.status} o tipo de contenido` )
  );
}
})
.then(currentUser => { // Cuando la promesa response.json() se resuelve
  displayUserInfo(currentUser); // hacer algo con el cuerpo analizado. })
  .catch(error => { // O si algo salió mal, simplemente registra el error. // Si el
  navegador del usuario está desconectado, el propio fetch() rechazará. // Si el servidor
  devuelve una mala respuesta entonces lanzamos un error arriba.
  console.log(`Error al recuperar el usuario actual: ${error}`);
});
```

La promesa devuelta por fetch() resuelve un objeto Response. La propiedad de estado de este objeto es el código de estado HTTP,

como 200 para las solicitudes exitosas o 404 para las respuestas "No encontradas".

(statusText da el texto estándar en inglés que acompaña al código de estado numérico). Convenientemente, la propiedad ok de una Respuesta es verdadera si el estado es 200 o cualquier código entre 200 y 299 y es falsa para cualquier otro código.

fetch() resuelve su promesa cuando la respuesta del servidor comienza a llegar, tan pronto como el estado HTTP y las cabeceras de respuesta están disponibles, pero normalmente antes de que el cuerpo completo de la respuesta haya llegado. Aunque el cuerpo no esté disponible todavía, puede examinar las cabeceras en este segundo paso del proceso de obtención. La propiedad headers de un objeto Response es un objeto Headers. Utilice su método has() para comprobar la presencia de una cabecera, o utilice su método get() para obtener el valor de una cabecera. Los nombres de las cabeceras HTTP no distinguen entre mayúsculas y minúsculas, por lo que puede pasar nombres de cabeceras en minúsculas o mixtas a estas funciones.

El objeto Headers también es iterable si alguna vez necesitas hacerlo:

```
fetch(url). then(response => { for(let [name,value] of response.headers) {  
  console.log(`#${name}: ${value}`);  
});
```

Si un servidor web responde a la petición de fetch(), la promesa devuelta se cumplirá con un objeto Response, incluso si la respuesta del servidor es un error 404 Not Found o un 500 Internal Server Error. fetch() sólo rechaza la promesa devuelta si no puede

contactar con el servidor web. Esto puede ocurrir si el ordenador del usuario está desconectado, el servidor no responde, o la URL especifica un nombre de host que no existe. Debido a que estas cosas pueden ocurrir en cualquier petición de red, siempre es una buena idea incluir una cláusula `.catch()` cada vez que se haga una llamada a `fetch()`.

## AJUSTE DE LOS PARÁMETROS DE LA SOLICITUD

A veces se desea pasar parámetros adicionales junto con la URL cuando se hace una petición. Esto puede hacerse añadiendo pares nombre/valor al final de una URL después de un `?`  Las clases `URL` y `URLSearchParams` (que se trataron en [§11.9](#)) facilitan la construcción de URLs de esta forma, y la función `fetch()` acepta objetos URL como su primer argumento, por lo que se pueden incluir parámetros de petición en una petición `fetch()` como ésta:

```
async function search(term) { let url = new URL("/api/search"); url.searchParams.set("q", term); let response = await fetch(url); if (!response.ok) throw new Error(response.statusText); let resultsArray = await response.json(); return resultsArray; }
```

## CONFIGURACIÓN DE LAS CABECERAS DE LAS SOLICITUDES

A veces es necesario establecer cabeceras en las peticiones `fetch()`. Por ejemplo, si está haciendo peticiones a la API web que requieren credenciales, puede que necesite incluir una cabecera de autorización que contenga esas credenciales. Para ello, puede utilizar la versión de dos argumentos de `fetch()`. Como antes, el primer argumento es una cadena o un objeto URL que especifica la URL a recuperar. El segundo argumento es un objeto que puede

proporcionar opciones adicionales, incluyendo las cabeceras de la solicitud:

```
let authHeaders = new Headers();
// No utilice Basic auth a menos que sea a través de una conexión HTTPS.
authHeaders.set("Authorization", `Basic ${btoa(`${username}:${password}`)}`);
fetch("/api/users/", { headers: authHeaders })
  .then(response => response.json())           // Se omite el manejo de errores...
then(usersList => displayAllUsers(usersList));
```

Hay un número de otras opciones que pueden ser especificadas en el segundo argumento de fetch(), y lo veremos de nuevo más adelante. Una alternativa a pasar dos argumentos a fetch() es pasar los mismos dos argumentos al constructor de Request() y luego pasar el objeto Request resultante a fetch():

```
let request = new Request(url, { headers });
fetch(request).then(response => ...);
```

## ANÁLISIS DE LOS CUERPOS DE RESPUESTA

En el proceso de tres pasos de fetch() que hemos demostrado, el segundo paso termina llamando a los métodos json() o text() del objeto Response y devolviendo el objeto Promise que esos métodos devuelven. Luego, el tercer paso comienza cuando esa Promesa se resuelve con el cuerpo de la respuesta analizado como un objeto JSON o simplemente como una cadena de texto.

Estos son probablemente los dos escenarios más comunes, pero no son las únicas formas de obtener el cuerpo de la respuesta de un servidor web. Además de json() y text(), el objeto Response también tiene estos métodos:

*arrayBuffer()*

Este método devuelve una Promise que resuelve un ArrayBuffer. Esto es útil cuando la respuesta contiene datos binarios. Puedes utilizar el ArrayBuffer para crear un array tipado ([§11.2](#)) o un objeto DataView ([§11.2.5](#)) del que puedas leer los datos binarios.

### *blob()*

Este método devuelve una Promise que resuelve un objeto Blob. Los Blobs no se tratan con detalle en este libro, pero su nombre significa "Binary Large Object", y son útiles cuando se esperan grandes cantidades de datos binarios. Si pides el cuerpo de la respuesta como un Blob, la implementación del navegador puede transmitir los datos de la respuesta a un archivo temporal y luego devolver un objeto Blob que representa ese archivo temporal. Los objetos Blob, por lo tanto, no permiten el acceso aleatorio al cuerpo de la respuesta de la forma en que lo hace un ArrayBuffer. Una vez que tengas un Blob, puedes crear una URL que haga referencia a él con `URL.createObjectURL()`, o puede utilizar la función FileReader API para obtener asincrónicamente el contenido del Blob como una cadena o un ArrayBuffer. En el momento de escribir este artículo, algunos navegadores también definen métodos `text()` y `arrayBuffer()` basados en promesas que ofrecen una ruta más directa para obtener el contenido de un Blob.

### *formData()*

Este método devuelve una Promise que resuelve un objeto FormData. Debe utilizar este método si espera que el cuerpo de la respuesta esté codificado en formato "multipart/form-data". Este formato es común en las solicitudes POST realizadas a un servidor, pero poco común en las respuestas del servidor, por lo que este método no se utiliza con frecuencia.

## **CUERPOS DE RESPUESTA DE STREAMING**

Además de los cinco métodos de respuesta que devuelven de forma asíncrona algún tipo de cuerpo de respuesta completo,

también hay una opción para transmitir el cuerpo de la respuesta, que es útil si hay algún tipo de procesamiento que puede hacer en los trozos del cuerpo de la respuesta a medida que llegan a través de la red. Pero la transmisión de la respuesta también es útil si quieras mostrar una barra de progreso para que el usuario pueda ver el progreso de la descarga.

La propiedad body de un objeto Response es un objeto ReadableStream. Si ya ha llamado a un método de respuesta como text() o json() que lee, analiza y devuelve el cuerpo, entonces bodyUsed será true para indicar que el flujo del cuerpo ya ha sido leído. Sin embargo, si bodyUsed es falso, entonces el flujo aún no ha sido leído. En este caso, puedes llamar a getReader() sobre response.body para obtener un objeto lector de flujos, y luego utilizar el método read() de este objeto lector para leer asíncronamente trozos de texto del flujo. El método read() devuelve una Promise que resuelve un objeto con las propiedades done y value. done será true si se ha leído todo el cuerpo o si se ha cerrado el flujo. Y value será el siguiente trozo, como un Uint8Array, o undefined si no hay más trozos.

Esta API de streaming es relativamente sencilla si se utiliza async y await, pero es sorprendentemente compleja si se intenta utilizar con Promesas en bruto. [El Ejemplo 15-10](#) demuestra la API definiendo una función streamBody(). Suponga que quiere descargar un archivo grande JSON y reportar el progreso de la descarga al usuario. No se puede hacer eso con el método json() del objeto Response, pero se podría hacer con la función streamBody(), así (asumiendo que se

define una función updateProgress() para establecer el atributo value en un elemento HTML <progress>):

```
fetch('big.json') . then(response => streamBody(response, updateProgress))
  . then(bodyText => JSON.parse(bodyText))
  . then(handleBigJSONObject);
```

La función streamBody() puede ser implementada como se muestra en el [Ejemplo 15-10](#).

*Ejemplo 15-10. Transmisión del cuerpo de la respuesta de una petición fetch()*

---

```
/** 
 * Una función asíncrona para transmitir el cuerpo de un
Objeto de respuesta
 *
 * obtenido de una petición fetch(). Pase el objeto Response como el primer
 * seguido de dos llamadas de retorno opcionales.
 *
 * Si especifica una función como segundo argumento, ese reportProgress
 * se llamará una vez por cada trozo que se reciba. La primera
 * es el número total de bytes recibidos hasta el momento. El segundo
 * es un número entre 0 y 1 que especifica lo completa que es la descarga
 * es. Sin embargo, si el objeto Response no tiene una cabecera "Content-Length",
entonces
 * este segundo argumento será siempre NaN.
 *
 * Si desea procesar los datos en trozos a medida que llegan, especifique un
 * como tercer argumento. Los trozos se pasarán, como Uint8Array
 * a este callback de processChunk.
 *
 * streamBody() devuelve una Promise que resuelve una cadena. Si
```

un trozo de proceso

- \* entonces esta cadena es la concatenación de los valores
- \* devuelto por esa llamada de retorno. En caso contrario, la cadena es la concatenación de
- \* los valores del chunk convertidos a cadenas UTF-8.

```
*/async function streamBody(response, reportProgress, processChunk) { // Cuántos bytes esperamos, o NaN si no hay cabecera
let expectedBytes = parseInt(response.headers.get("ContentLength"));
let bytesRead = 0; // Cuántos bytes se han recibido hasta ahora
let reader = response.body.getReader(); // Leer los bytes con esta función
let decoder = new TextDecoder("utf-8"); // Para convertir los bytes en texto
let body = ""; // Texto leído hasta el momento

while(true) { // Bucle hasta que salgamos por debajo
let {done, value} = await reader.read(); // Leer un trozo

if (value) { // Si tenemos un array de bytes:
if (processChunk) { // Procesa los bytes si
let processed = processChunk(value); // se ha pasado una callback.
if (processed) { body += processed; }
} else { // De lo contrario, convierte los bytes
body += decoder.decode(value, {stream: true}); // a texto.
}
}

if (reportProgress) { // Si se ha realizado una llamada de retorno al progreso
bytesRead += valor.longitud; // pasado, entonces llámalo
reportProgress(bytesRead, bytesRead / expectedBytes); }
} if (done) { // Si este es el último trozo, break; // salir del bucle
break;
}
```

```
return body; // Devuelve el texto del cuerpo que hemos acumulado }
```

Esta API de streaming es nueva en el momento de escribir este artículo y se espera que evolucione. En particular, hay planes para hacer que los objetos ReadableStream sean iterables de forma asíncrona para que puedan ser utilizados con bucles for/await ([§13.4.1](#)).

## ESPECIFICANDO EL MÉTODO Y EL CUERPO DE LA SOLICITUD

En cada uno de los ejemplos de `fetch()` mostrados hasta ahora, hemos realizado una petición GET HTTP (o HTTPS). Si desea utilizar un método de solicitud diferente (como POST, PUT o DELETE), simplemente utilice la versión de dos argumentos de `fetch()`, pasando un objeto Options con un parámetro de método:

```
fetch(url, { method: "POST" }).then(r =>
r.json()).then(handleResponse);
```

Las peticiones POST y PUT suelen tener un cuerpo de petición que contiene los datos que se enviarán al servidor. Siempre que la propiedad del método no esté establecida como "GET" o "HEAD" (que no admiten cuerpos de petición), puede especificar un cuerpo de petición estableciendo la propiedad `body` del objeto Options:

```
fetch(url, { method: "POST",
body: "hello world" })
```

Cuando se especifica un cuerpo de solicitud, el navegador añade automáticamente una cabecera "Content-Length" apropiada a la solicitud. Cuando el cuerpo es una cadena, como en el ejemplo anterior, el navegador establece por defecto la cabecera

"Content-Type" como "text/plain;charset=UTF-8". Es posible que tenga que anular este valor por defecto si especifica un cuerpo de cadena de algún tipo más específico como "text/html" o "application/json":

```
fetch(url, { method: "POST", headers: new  
Headers({"Content-Type": "application/json"}), body:  
JSON.stringify(requestBody) })
```

La propiedad body del objeto fetch() options no tiene por qué ser una cadena. Si tiene datos binarios en un array tipificado o un objeto DataView o un ArrayBuffer, puede establecer la propiedad body a ese valor y especificar una cabecera "Content-Type" apropiada. Si tiene datos binarios en forma de Blob, puede simplemente establecer body al Blob. Los Blobs tienen una propiedad type que especifica su tipo de contenido, y el valor de esta propiedad se utiliza como valor por defecto de la cabecera "Content-Type".

Con las peticiones POST, es algo común pasar un conjunto de parámetros de nombre/valor en el cuerpo de la petición (en lugar de codificarlos en la parte de la consulta de la URL). Hay dos maneras de hacerlo:

- Puede especificar los nombres y valores de sus parámetros con URLSearchParams (que vimos anteriormente en esta sección, y que está documentado en §11.9) y luego pasar el URLSearchParams como valor de la propiedad body. Si hace esto, el cuerpo se establecerá como una cadena que se parece a la porción de consulta de una URL, y la cabecera "Content-Type" se establecerá automáticamente como "application/x-www-form-urlencoded;charset=UTF-8".

- Si, en cambio, especifica los nombres y valores de sus parámetros con un objeto FormData, el cuerpo utilizará una codificación multiparte más verbose y "Content-Type" se establecerá como "multipart/form-data; boundary=..." con una cadena de límite única que coincide con el cuerpo. El uso de un objeto FormData es especialmente útil cuando los valores que se quieren subir son largos, o son objetos File o Blob que pueden tener cada uno su propio "Content-Type". Los objetos FormData se pueden crear e inicializar con valores pasando un elemento <form> al constructor FormData(). Pero también se pueden crear cuerpos de solicitud "multipart/form-data" invocando el constructor FormData() sin argumentos e inicializando los pares nombre/valor que representa con los métodos set() y append().

## CARGA DE ARCHIVOS CON FETCH()

La carga de archivos desde el ordenador de un usuario a un servidor web es una tarea común y puede llevarse a cabo utilizando un objeto FormData como cuerpo de la petición. Una forma común de obtener un objeto File es mostrar un elemento <input type="file"> en su página web y escuchar los eventos "change" en ese elemento. Cuando se produce un evento de "cambio", la matriz de archivos del elemento de entrada debe contener al menos un objeto File. Los objetos de archivo también están disponibles a través de la API de arrastrar y soltar de HTML. Esta API no se trata en este libro, pero puedes obtener archivos de la matriz dataTransfer.files del objeto de evento pasado a un oyente de eventos "drop".

Recuerda también que los objetos File son un tipo de Blob, y a veces puede ser útil cargar Blobs. Suponga que ha escrito una aplicación web que permite al usuario crear dibujos en un

elemento <canvas>. Puedes subir los dibujos del usuario como archivos PNG con un código como el siguiente:

```
// La función canvas.toBlob() está basada en un callback.  
// Esta es una envoltura basada en Promise para ello.  
async function getCanvasBlob(canvas) { return new  
Promise((resolve, reject) => { canvas.toBlob(resolve); });  
}  
  
// Así es como subimos un archivo PNG desde un canvas async function  
uploadCanvasImage(canvas) { let pngblob = await getCanvasBlob(canvas); let  
formdata = new FormData(); formdata.set("canvasimage", pngblob); let response =  
await fetch("/upload", { method: "POST", body: formdata }); let body = await  
response.json();  
}
```

## SOLICITUDES DE ORIGEN CRUZADO

La mayoría de las veces, fetch() es utilizada por las aplicaciones web para solicitar datos de su propio servidor web. Este tipo de solicitudes se conocen como solicitudes del mismo origen porque la URL que se pasa a fetch() tiene el mismo origen (protocolo más nombre de host más puerto) que el documento que contiene el script que realiza la solicitud.

Por razones de seguridad, los navegadores web generalmente no permiten (aunque hay excepciones para las imágenes y los scripts) las solicitudes de red de origen cruzado. Sin embargo, el intercambio de recursos entre orígenes, o CORS, permite realizar peticiones seguras entre orígenes. Cuando se utiliza fetch() con una URL de origen cruzado, el navegador añade una cabecera "Origin" a la petición (y no permite que se anule mediante la propiedad headers) para notificar al servidor web que la petición

procede de un documento con un origen diferente. Si el servidor responde a la petición con una cabecera "Access-Control-Allow-Origin" apropiada, entonces la petición procede. En caso contrario, si el servidor no permite explícitamente la petición, entonces la promesa devuelta por `fetch()` es rechazada.

## ABORTAR UNA SOLICITUD

A veces puede querer abortar una petición `fetch()` que ya ha emitido, tal vez porque el usuario ha pulsado un botón de cancelación o la petición está tardando demasiado. La API de `fetch` permite abortar las peticiones mediante las clases `AbortController` y `AbortSignal`. (Estas clases definen un mecanismo de interrupción genérico que puede ser utilizado también por otras APIs).

Si quiere tener la opción de abortar una petición `fetch()`, entonces cree un objeto `AbortController` antes de iniciar la petición. La propiedad `signal` del objeto controlador es un objeto `AbortSignal`. Pase este objeto señal como el valor de la propiedad `signal` del objeto `options` que pasa a `fetch()`. Una vez hecho esto, puede llamar al método `abort()` del objeto controlador para abortar la solicitud, lo que hará que cualquier objeto `Promise` relacionado con la solicitud de obtención sea rechazado con una excepción.

Este es un ejemplo del uso del mecanismo `AbortController` para imponer un tiempo de espera para las solicitudes de obtención:



```
// Esta función es como fetch(), pero añade soporte para un tiempo de espera
// propiedad en el objeto options y aborta la obtención si no se completa // en el
// número de milisegundos especificado por esa propiedad.
function fetchWithTimeout(url, options={}) { if (options.timeout) { // Si la propiedad
timeout existe y es distinta de cero let controller = new AbortController(); // Crear un
controlador options.signal = controller.signal; // Establecer la propiedad signal
// Iniciar un temporizador que enviará la señal de interrupción después de la
// número de milisegundos han pasado. Tenga en cuenta que nunca cancelamos
// este temporizador. Llamar a abort() después de que se complete la
obtención no tiene // ningún efecto. setTimeout(() => { controller.abort(); },
options.timeout);
} // Ahora sólo hay que realizar una búsqueda
normal return fetch(url, options);
}
```

## OPCIONES DE SOLICITUD DIVERSAS

Hemos visto que se puede pasar un objeto Options como segundo argumento a `fetch()` (o como segundo argumento al constructor de `Request()`) para especificar el método de petición, las cabeceras de la petición y el cuerpo de la misma. También admite otras opciones, como las siguientes:

### *caché*

Utilice esta propiedad para anular el comportamiento de almacenamiento en caché por defecto del navegador. El almacenamiento en caché de HTTP es un tema complejo que está más allá del alcance de este libro, pero si sabes algo sobre

su funcionamiento, puedes utilizar los siguientes valores legales de cache:

#### *"por defecto"*

Este valor especifica el comportamiento de la caché por defecto. Las respuestas frescas en la caché se sirven directamente desde la caché, y las respuestas antiguas se revalidan antes de ser servidas.

#### *"no-store"*

Este valor hace que el navegador ignore su caché. La caché no se comprueba en busca de coincidencias cuando se realiza la solicitud y no se actualiza cuando llega la respuesta.

#### *"recarga"*

Este valor le dice al navegador que siempre haga una petición de red normal, ignorando la caché. Sin embargo, cuando la respuesta llega, se almacena en la caché.

#### *"no-cache"*

Este valor (de nombre engañoso) indica al navegador que no sirva valores frescos de la caché. Los valores frescos o antiguos de la caché se revalidan antes de ser devueltos.

#### *"forzar el caché"*

Este valor le dice al navegador que sirva las respuestas de la caché incluso si son obsoletas.

#### *redirigir*

Esta propiedad controla cómo el navegador maneja las respuestas de redirección del servidor. Los tres valores legales son:

#### *"seguir"*

Este es el valor por defecto, y hace que el navegador siga las redirecciones automáticamente. Si utiliza este valor por defecto,

los objetos Response que obtenga con fetch() nunca deberían tener un estado en el rango de 300 a 399.

*"error"*

Este valor hace que fetch() rechace su promesa devuelta si el servidor devuelve una respuesta de redirección.

*"manual"*

Este valor significa que quiere manejar manualmente las respuestas de redirección, y la Promesa devuelta por fetch() puede resolver un objeto Response con un estado en el rango 300 a 399. En este caso, tendrá que utilizar la cabecera "Location" de la Respuesta para seguir manualmente la redirección.

*referente*

Puede establecer esta propiedad a una cadena que contenga una URL relativa para especificar el valor de la cabecera HTTP "Referer" (que históricamente se escribe mal con tres erres en lugar de cuatro). Si establece esta propiedad como una cadena vacía, el encabezado "Referer" se omitirá en la solicitud.

### **15.11.2 Eventos enviados por el servidor**

Una característica fundamental del protocolo HTTP en el que se basa la web es que los clientes inician las peticiones y los servidores responden a ellas. Sin embargo, a algunas aplicaciones web les resulta útil que su servidor les envíe notificaciones cuando se producen eventos. Esto no es algo natural en HTTP, pero la técnica que se ha ideado es que el cliente haga una petición al servidor y luego ni el cliente ni el servidor cierren la conexión. Cuando el servidor tiene algo que comunicar al cliente, escribe datos en la conexión pero la mantiene abierta. El efecto es como si el cliente hiciera una petición de red y el servidor respondiera de

forma lenta y a ráfagas, con pausas significativas entre las ráfagas de actividad. Las conexiones de red como ésta no suelen permanecer abiertas para siempre, pero si el cliente detecta que la conexión se ha cerrado, puede simplemente hacer otra petición para reabrir la conexión.

Esta técnica para permitir que los servidores envíen mensajes a los clientes es sorprendentemente efectiva (aunque puede ser costosa en el lado del servidor porque éste debe mantener una conexión activa con todos sus clientes). Dado que es un patrón de programación útil, JavaScript del lado del cliente lo soporta con la API EventSource. Para crear este tipo de conexión de petición de larga duración a un servidor web, simplemente pasa una URL al constructor EventSource(). Cuando el servidor escribe datos (con el formato adecuado) en la conexión, el objeto EventSource los traduce en eventos que puedes escuchar:

```
let ticker = new EventSource("stockprices.php"); ticker.  
addEventListener("bid", (event) => { displayNewBid(event. data); })
```

El objeto de evento asociado con un evento de mensaje tiene una propiedad data que contiene la cadena que el servidor envió como carga útil para este evento.

El objeto de evento también tiene una propiedad type, como todos los objetos de evento, que especifica el nombre del evento. El servidor determina el tipo de los eventos que se generan. Si el servidor omite un nombre de evento en los datos que escribe, entonces el tipo de evento es por defecto "mensaje".

El protocolo de eventos enviados por el servidor es sencillo. El cliente inicia una conexión con el servidor (cuando crea el objeto EventSource), y el servidor mantiene esta conexión abierta.

Cuando ocurre un evento, el servidor escribe líneas de texto en la conexión. Un evento que pasa por el cable podría tener este aspecto, si se omitieran los comentarios:

```
evento: bid // establece el tipo del objeto de evento
data: GOOG //
establece la propiedad data
data: 999 // añade una nueva línea y más
datos // una línea en blanco marca el final del evento
```

Hay algunos detalles adicionales en el protocolo que permiten que los eventos reciban IDs y que un cliente que se reconecta le diga al servidor cuál fue el ID del último evento que recibió, para que un servidor pueda reenviar cualquier evento que haya perdido. Sin embargo, estos detalles son invisibles en el lado del cliente y no se discuten aquí.

Una aplicación obvia para los Eventos Enviados por el Servidor es para colaboraciones multiusuario como el chat en línea. Un cliente de chat podría usar fetch() para enviar mensajes a la sala de chat y suscribirse al flujo de charlas con un objeto EventSource. [El Ejemplo 15-11](#) demuestra lo fácil que es escribir un cliente de chat como este con EventSource.

#### *Ejemplo 15-11. Un simple cliente de chat usando EventSource*

---

```
<html>
<head><title>Chat de SSE</title></head>
<cuerpo>
```

La interfaz del chat es sólo un campo de entrada de texto.

```

<!-- Los nuevos mensajes de chat se insertarán antes de este campo de entrada
-->
<input id="input" style="width:100%; padding:10px; border:solid black 2px"/>
<script>
// Ocuparse de algunos detalles de la interfaz de usuario
let nick = prompt("Introduce tu apodo"); // Obtener el apodo del usuario let input =
document.getElementById("input"); // Encontrar el campo de entrada input.focus(); // Establecer el foco del teclado

// Regístrese para la notificación de nuevos mensajes utilizando EventSource let chat = new
EventSource("/chat"); chat.addEventListener("chat", event => { // Cuando llega un mensaje
de chat let div = document.createElement("div"); // Cree un <div> div.append(event.data);
// Añada el texto del mensaje input.before(div); // Y añada el div antes del input
input.scrollIntoView(); // Asegúrese de que el input de entrada es visible });

// Publicar los mensajes del usuario en el servidor utilizando fetch
input.addEventListener("change", ()=>{ // Cuando el usuario hace un strike
return fetch("/chat", { // Iniciar una petición HTTP a esta url.
método: "POST", // Que sea una petición POST con body body: nick + ":" +
input.value // que se ajuste al nick del usuario y al input.
})
.catch(e => console.error); // Ignorar la respuesta, pero registrar cualquier error.
input.value = ""; // Borrar la entrada
});
</script>
</body>
</html>
```

El código del lado del servidor para este programa de chat no es mucho más complicado que el código del lado del cliente. El Ejemplo [15-12](#) es un simple servidor HTTP de Node. Cuando un cliente solicita la URL raíz "/", envía el código del cliente de chat

mostrado en el [Ejemplo 15-11](#). Cuando un cliente hace una solicitud GET para la URL "/chat", guarda el objeto de respuesta y mantiene esa conexión abierta. Y cuando un cliente hace una petición POST a "/chat", usa el cuerpo de la petición como un mensaje de chat y lo escribe, usando el formato "text/event-stream" a cada uno de los objetos de respuesta guardados. El código del servidor escucha en el puerto 8080, así que después de ejecutarlo con Node, apunta tu navegador a <http://localhost:8080> para conectarte y empezar a chatear contigo mismo.

*Ejemplo 15-12. Un servidor de chat de eventos enviados por el servidor*

---

```
// Esto es JavaScript del lado del servidor, pensado para ser ejecutado con NodeJS.  
// Implementa una sala de chat muy simple y completamente anónima.  
// POST nuevos mensajes a /chat, o GET un flujo de texto/eventos de mensajes  
// desde la misma URL. Hacer una petición GET a / devuelve un simple archivo  
HTML // que contiene la interfaz de usuario del chat del lado del cliente. const  
http = require("http"); const fs = require("fs"); const url = require("url");  
  
// El archivo HTML para el cliente de chat. Se utiliza a continuación. const  
clientHTML = fs.readFileSync("chatClient.html");  
  
// Un array de objetos ServerResponse a los que vamos a enviar eventos let clients = [];  
  
// Crear un nuevo servidor, y escuchar en el puerto 8080.  
// Conéctese a http://localhost:8080/ para utilizarlo. let server = new  
http.Server(); server.listen(8080);  
  
// Cuando el servidor recibe una nueva solicitud, ejecuta esta función server.on("request",  
(request, response) => {
```

```

// Analizar la URL solicitada let pathname = url.parse(request.url).
pathname;

// Si la petición era para "/", envía el chat del lado del cliente
UI. if (pathname === "/") { // Una solicitud de respuesta de la UI del chat.
writeHead(200, {"Content-Type": "text/html"}). end(clientHTML);

}

// De lo contrario, envía un error 404 para cualquier ruta que no sea
"/chat" o para
// cualquier método que no sea "GET" y "POST" else if (pathname !== "/chat" ||
(request.method !== "GET" && request.method !== "POST")) { response.
writeHead(404). end(); }

}

// Si la solicitud de /chat fue un GET, entonces un cliente se está conectando.
else if (request.method === "GET") { acceptNewClient(request, response);
}

// En caso contrario, la petición /chat es un POST de un nuevo mensaje else {
broadcastNewMessage(request, response); }

});

// Esto maneja las solicitudes GET para el punto final /chat que se generan cuando
// el cliente crea un nuevo objeto EventSource (o cuando el
EventSource
// se reconecta automáticamente). function
acceptNewClient(request, response) {
    // Recordar el objeto respuesta para poder enviarle futuros mensajes clientes.
push(response);

    // Si el cliente cierra la conexión, elimina el objeto // de respuesta
correspondiente del array de clientes activos request.connection.on("end", () => { clients.splice(clients.indexOf(response), 1);
}

```

```
response.end(); });

// Establece las cabeceras y envía un evento de chat inicial a esta única respuesta del cliente. writeHead(200, { "Content-Type": "text/event-stream",
  "Conexión": "keep-alive",
  "Cache-Control": "no-cache" }); response.write("event: chat\n"data:
Connected\n\n");

// Observa que aquí no llamamos intencionadamente a response.end(). // Mantener la conexión abierta es lo que hace que los Eventos Enviados por el Servidor funcionen.
}

// Esta función se llama en respuesta a las peticiones POST al /punto final del chat
// que los clientes envían cuando los usuarios escriben un nuevo mensaje. async
function broadcastNewMessage(request, response) {
  // En primer lugar, lee el cuerpo de la solicitud para obtener el mensaje del usuario
request.setEncoding("utf8"); let body = ""; for await (let chunk of request) { body += chunk; }

  // Una vez que hayamos leído el cuerpo envía una respuesta vacía y cierra la conexión response.writeHead(200).end();

  // Formatear el mensaje en formato de texto/flujo de eventos, prefijando cada // línea con "data: " let message = "data: " + body.replace("\n", "\ndata: ");

  // Dar a los datos del mensaje un prefijo que lo defina como Evento "chat"
// y darle un sufijo de doble línea nueva que marque el final del evento.
let event = `event: chat\n${mensaje}\n`;

// Ahora envía este evento a todos los clientes que están escuchando clients. forEach(client => client.write(event)); }
```

### 15.11.3 WebSockets

La API WebSocket es una interfaz sencilla para un protocolo de red complejo y potente. Los WebSockets permiten que el código JavaScript del navegador intercambie fácilmente mensajes de texto y binarios con un servidor. Al igual que con los eventos enviados por el servidor, el cliente debe establecer la conexión, pero una vez establecida la conexión, el servidor puede enviar mensajes al cliente de forma asíncrona. A diferencia de SSE, los mensajes binarios están soportados, y los mensajes pueden ser enviados en ambas direcciones, no sólo del servidor al cliente.

El protocolo de red que permite los WebSockets es una especie de extensión de HTTP. Aunque la API de WebSocket recuerda a los sockets de red de bajo nivel, los puntos finales de conexión no se identifican por dirección IP y puerto. En su lugar, cuando se quiere conectar a un servicio mediante el protocolo WebSocket, se especifica el servicio con una URL, igual que se haría con un servicio web. Sin embargo, las URL de WebSocket comienzan con `wss://` en lugar de `https://`. (Los navegadores suelen restringir los WebSockets para que sólo funcionen en páginas cargadas a través de conexiones seguras `https://`).

Para establecer una conexión WebSocket, el navegador establece primero una conexión HTTP y envía al servidor una cabecera `Upgrade: websocket` solicitando que la conexión pase del protocolo HTTP al protocolo WebSocket. Lo que esto significa es que para utilizar WebSockets en tu JavaScript del lado del cliente, necesitarás estar trabajando con un servidor web que también hable el protocolo WebSocket, y necesitarás tener código del lado del servidor escrito para enviar y recibir datos utilizando ese

protocolo. Si tu servidor está configurado de esa manera, entonces esta sección explicará todo lo que necesitas saber para manejar el extremo del cliente de la conexión. Si su servidor no soporta el protocolo WebSocket, considere el uso de Eventos Enviados por el Servidor ([§15.11.2](#)) en su lugar.

## CREAR, CONECTAR Y DESCONECTAR WEBSOCKETS

Si desea comunicarse con un servidor habilitado para WebSocket, cree un objeto WebSocket, especificando la URL `wss://` que identifica el servidor y el servicio que desea utilizar: `let socket = new WebSocket("wss://ejemplo.com/stockticker");`

Cuando se crea un WebSocket, el proceso de conexión comienza automáticamente. Pero un WebSocket recién creado no se conectará cuando se devuelva por primera vez.

La propiedad `readyState` del socket especifica en qué estado se encuentra la conexión. Esta propiedad puede tener los siguientes valores:

*WebSocket.CONNECTING*

Este WebSocket se está conectando.

*WebSocket.OPEN*

Este WebSocket está conectado y listo para la comunicación.

*WebSocket.CLOSING*

Esta conexión WebSocket se está cerrando.

*WebSocket.CERRADO*

Este WebSocket ha sido cerrado; no es posible ninguna otra comunicación. Este estado también puede ocurrir cuando el intento de conexión inicial falla.

Cuando un WebSocket pasa del estado CONNECTING al estado OPEN, lanza un evento "open", y usted puede escuchar este evento estableciendo la propiedad onopen del WebSocket o llamando a addEventListener() en ese objeto.

Si se produce un error de protocolo o de otro tipo en una conexión WebSocket, el objeto WebSocket lanza un evento "error". Puede establecer onerror para definir un controlador o, alternativamente, utilizar addEventListener().

Cuando hayas terminado con un WebSocket, puedes cerrar la conexión llamando al método close() del objeto WebSocket. Cuando un WebSocket cambia al estado CLOSED, dispara un evento "close", y puedes establecer la propiedad onclose para escuchar este evento.

## ENVÍO DE MENSAJES A TRAVÉS DE UN WEBSOCKET

Para enviar un mensaje al servidor en el otro extremo de una conexión WebSocket, basta con invocar el método send() del objeto WebSocket. send() espera un único argumento de mensaje, que puede ser una cadena, un Blob, un ArrayBuffer, un array tipado o un objeto DataView.

El método send() almacena en el búfer el mensaje especificado para ser transmitido y devuelve antes de que el mensaje sea realmente enviado. La propiedad bufferedAmount del objeto WebSocket especifica el número de bytes que están almacenados

en el buffer pero que aún no han sido enviados.

(Sorprendentemente, los WebSockets no disparan ningún evento cuando este valor llega a 0).

## RECIBIR MENSAJES DE UN WEBSOCKET

Para recibir mensajes de un servidor a través de un WebSocket, registre un manejador de eventos para los eventos "mensaje", ya sea estableciendo la propiedad onmessage del objeto WebSocket, o llamando a

addEventListener(). El objeto asociado a un evento "mensaje" es una instancia de MessageEvent con una propiedad data que contiene el mensaje del servidor. Si el servidor envió texto codificado en UTF-8, entonces event.data será una cadena que contenga ese texto.

Si el servidor envía un mensaje que consiste en datos binarios en lugar de texto, entonces la propiedad data será (por defecto) un objeto Blob que representa esos datos. Si prefiere recibir mensajes binarios como ArrayBuffers en lugar de Blobs, establezca la propiedad binaryType de la propiedad objeto WebSocket a la cadena "arraybuffer".

Hay una serie de APIs web que utilizan objetos MessageEvent para intercambiar mensajes. Algunas de estas APIs utilizan el algoritmo de clonación estructurada (véase "["El algoritmo de clonación estructurada"](#)") para permitir estructuras de datos complejas como carga útil del mensaje. WebSockets no es una de esas APIs: los mensajes que se intercambian a través de un WebSocket son una única cadena de caracteres Unicode o una única cadena de bytes (representada como un Blob o un ArrayBuffer).

## NEGOCIACIÓN DEL PROTOCOLO

El protocolo WebSocket permite el intercambio de mensajes de texto y binarios, pero no dice nada en absoluto sobre la estructura o el significado de esos mensajes. Las aplicaciones que utilizan WebSockets deben construir su propio protocolo de comunicación sobre este sencillo mecanismo de intercambio de mensajes. El uso de las URLs `wss://` ayuda a ello: cada URL suele tener sus propias reglas sobre cómo deben intercambiarse los mensajes. Si escribe código para conectarse a `wss://example.com/stockticker`, entonces probablemente sabrá que recibirá mensajes sobre los precios de las acciones.

Sin embargo, los protocolos tienden a evolucionar. Si un hipotético protocolo de cotización de acciones se actualiza, puede definir una nueva URL y conectarse al servicio actualizado como `wss://ejemplo.com/stockticker/v2`.

Sin embargo, el control de versiones basado en la URL no siempre es suficiente. Con protocolos complejos que han evolucionado a lo largo del tiempo, puedes acabar con servidores desplegados que soportan múltiples versiones del protocolo y clientes desplegados que soportan un conjunto diferente de versiones del protocolo.

Anticipándose a esta situación, el protocolo y la API de WebSocket incluyen una función de negociación del protocolo a nivel de aplicación. Cuando se llama al constructor de `WebSocket()`, la URL `wss://` es el primer argumento, pero también se puede pasar una matriz de cadenas como segundo argumento. Si haces esto, estás especificando una lista de protocolos de aplicación que sabes manejar y pidiendo al servidor que elija uno. Durante el proceso

de conexión, el servidor elegirá uno de los protocolos (o fallará con un error si no soporta ninguna de las opciones del cliente). Una vez establecida la conexión, la propiedad de protocolo del El objeto WebSocket especifica qué versión del protocolo ha elegido el servidor.

## 15.12 Almacenamiento

Las aplicaciones web pueden utilizar las API del navegador para almacenar datos localmente en el ordenador del usuario. Este almacenamiento del lado del cliente sirve para dotar al navegador web de memoria. Las aplicaciones web pueden almacenar las preferencias del usuario, por ejemplo, o incluso guardar su estado completo, de modo que puedan reanudar la actividad exactamente donde la dejó al final de su última visita. El almacenamiento en el lado del cliente está segregado por origen, de modo que las páginas de un sitio no pueden leer los datos almacenados por las páginas de otro sitio. Pero dos páginas del mismo sitio pueden compartir el almacenamiento y utilizarlo como mecanismo de comunicación. Los datos introducidos en un formulario de una página pueden mostrarse en una tabla de otra página, por ejemplo. Las aplicaciones web pueden elegir el tiempo de vida de los datos que almacenan: los datos pueden almacenarse temporalmente para que se conserven sólo hasta que se cierre la ventana o se salga del navegador, o pueden guardarse en el ordenador del usuario y almacenarse permanentemente para que estén disponibles meses o años después.

Hay varias formas de almacenamiento del lado del cliente:

## *Almacenamiento en la web*

La API de Almacenamiento Web consiste en los objetos `localStorage` y `sessionStorage`, que son esencialmente objetos persistentes que asignan claves de cadena a valores de cadena. Web Storage es muy fácil de usar y es adecuado para almacenar grandes (pero no enormes) cantidades de datos.

## *Galletas*

Las cookies son un viejo mecanismo de almacenamiento del lado del cliente que fue diseñado para ser utilizado por scripts del lado del servidor. Una complicada API de JavaScript hace que las cookies puedan ser utilizadas por los scripts del lado del cliente, pero son difíciles de usar y sólo son adecuadas para almacenar pequeñas cantidades de datos textuales. Además, todos los datos almacenados como cookies se transmiten siempre al servidor con cada petición HTTP, incluso si los datos sólo interesan al cliente.

## *IndexedDB*

IndexedDB es una API asíncrona para una base de datos de objetos que soporta la indexación.

### **ALMACENAMIENTO, SEGURIDAD Y PRIVACIDAD**

Los navegadores web a menudo ofrecen recordar las contraseñas web por ti, y las almacenan de forma segura en forma cifrada en el dispositivo. Pero ninguna de las formas de almacenamiento de datos del lado del cliente descritas en este capítulo implica el cifrado: debes asumir que cualquier cosa que tus aplicaciones web guarden reside en el dispositivo del usuario de forma no cifrada. Por lo tanto, los datos almacenados son accesibles para los usuarios curiosos que comparten el acceso al dispositivo y para el software malicioso (como el spyware) que existe en el dispositivo. Por esta razón, nunca debe utilizarse ninguna forma de almacenamiento en el lado del cliente para contraseñas, números de cuentas financieras u otra información sensible similar.

### **15.12.1 `localStorage` y `sessionStorage`**

Las propiedades `localStorage` y `sessionStorage` del objeto `Window` se refieren a objetos `Storage`. Un objeto `Storage` se

comporta de forma muy parecida a un objeto JavaScript normal, excepto que:

- Los valores de las propiedades de los objetos Storage deben ser cadenas.

Las propiedades almacenadas en un objeto Storage persisten. Si estableces una propiedad del objeto localStorage y luego el usuario recarga la página, el valor que guardaste en esa propiedad sigue estando disponible para tu programa.

Puedes utilizar el objeto localStorage de esta manera, por ejemplo:

```
let name = localStorage.username; // Consultar un valor almacenado.  
if (!nombre) { nombre = prompt("¿Cuál es su nombre?"); // Hacer una pregunta al  
usuario.  
localStorage.username = name; // Almacenar la respuesta del usuario. }
```

Puedes utilizar el operador delete para eliminar las propiedades de localStorage y sessionStorage, y puedes utilizar un bucle for/in o Object.keys() para enumerar las propiedades de un objeto Storage. Si quieres eliminar todas las propiedades de un objeto Storage, llama al método clear():

```
localStorage.clear();
```

Los objetos de almacenamiento también definen los métodos getItem(), setItem() y deleteItem(), que puedes utilizar en lugar del acceso directo a las propiedades y el operador de borrado si lo deseas.

Ten en cuenta que las propiedades de los objetos Storage sólo pueden almacenar cadenas. Si quieres almacenar y recuperar otro tipo de datos, tendrás que codificarlos y decodificarlos tú mismo.

Por ejemplo:

```
// Si almacena un número, se convierte automáticamente en una cadena. // No  
olvides parsearlo cuando lo recuperes del almacenamiento. localStorage.x = 10; let x  
= parseInt(localStorage.x);  
  
// Convertir una fecha en una cadena cuando se establece, y analizarla cuando se  
obtiene localStorage.lastRead = (new Date()).toUTCString(); let lastRead = new  
Date(Date.parse(localStorage.lastRead)); // JSON es una codificación conveniente  
para cualquier primitiva o estructura de datos localStorage.data = JSON.  
stringify(data); // Codificar y almacenar let data = JSON.parse(localStorage.data); //  
Recuperar y decodificar.
```

## VIDA ÚTIL Y ALCANCE DEL ALMACENAMIENTO

La diferencia entre localStorage y sessionStorage tiene que ver con la duración y el alcance del almacenamiento. Los datos almacenados a través de localStorage son permanentes: no caducan y permanecen almacenados en el dispositivo del usuario hasta que una aplicación web los elimina o el usuario pide al navegador (a través de alguna interfaz de usuario específica del navegador) que los elimine.

localStorage tiene como ámbito el origen del documento. Como se explica en "[La política del mismo origen](#)", el origen de un documento se define por su protocolo, nombre de host y puerto. Todos los documentos con el mismo origen comparten los mismos datos de localStorage (independientemente del origen de los scripts que realmente acceden a localStorage). Pueden leer los datos de los demás, y pueden sobrescribir los datos de los demás. Pero los documentos con diferentes orígenes nunca pueden leer o sobrescribir los datos del otro (incluso si ambos están ejecutando un script desde el mismo servidor de terceros).

Tenga en cuenta que localStorage también está limitado por la implementación del navegador. Si visitas un sitio con Firefox y luego vuelves a visitarlo con Chrome (por ejemplo), cualquier dato almacenado durante la primera visita no será accesible durante la segunda.

Los datos almacenados a través de sessionStorage tienen un tiempo de vida diferente al de los datos almacenados a través de localStorage: tienen el mismo tiempo de vida que la ventana de nivel superior o la pestaña del navegador en la que se está ejecutando el script que los almacenó. Cuando la ventana o la pestaña se cierran definitivamente, cualquier dato almacenado a través de sessionStorage se borra. (Sin embargo, ten en cuenta que los navegadores modernos tienen la capacidad de reabrir las pestañas cerradas recientemente y restaurar la última sesión de navegación, por lo que el tiempo de vida de estas pestañas y su sessionStorage asociado puede ser más largo de lo que parece).

Al igual que localStorage, sessionStorage tiene como ámbito el para que los documentos con diferentes orígenes nunca compartan sessionStorage. Pero el sessionStorage también tiene un alcance por ventana. Si un usuario tiene dos pestañas del navegador que muestran documentos del mismo origen, esas dos pestañas tienen datos sessionStorage separados: los scripts que se ejecutan en una pestaña no pueden leer o sobrescribir los datos escritos por los scripts en la otra pestaña, incluso si ambas pestañas están visitando exactamente la misma página y están ejecutando exactamente los mismos scripts.

## EVENTOS DE ALMACENAMIENTO

Cada vez que los datos almacenados en localStorage cambian, el navegador desencadena un evento "storage" en cualquier otro objeto Window en el que esos datos sean visibles (pero no en la ventana que realizó el cambio). Si un navegador tiene dos pestañas abiertas a páginas con el mismo origen, y una de esas páginas almacena un valor en localStorage, la otra pestaña recibirá un evento "storage".

Registra un controlador para los eventos de "almacenamiento", ya sea estableciendo window.onstorage o llamando a window.addEventListener() con el tipo de evento "almacenamiento".

El objeto de evento asociado a un evento de "almacenamiento" tiene algunas propiedades importantes:

### *clave*

El nombre o la clave del elemento que fue establecido o eliminado. Si se llamó al método clear(), esta propiedad será nula.

### *newValue*

Contiene el nuevo valor del elemento, si lo hay. Si se llamó a removeItem(), esta propiedad no estará presente.

### *oldValue*

Mantiene el valor antiguo de un elemento existente que cambió o fue eliminado. Si se añade una nueva propiedad (sin valor antiguo), esta propiedad no estará presente en el objeto de evento.

### *storageArea*

El objeto Storage que ha cambiado. Normalmente es el objeto localStorage.

*url*

La URL (como una cadena) del documento cuyo script hizo este cambio de almacenamiento.

Tenga en cuenta que localStorage y el evento "storage" pueden servir como un mecanismo de difusión por el cual un navegador envía un mensaje a todas las ventanas que están visitando actualmente el mismo sitio web. Si un usuario solicita que un sitio web deje de realizar animaciones, por ejemplo, el sitio podría almacenar esa preferencia en localStorage para poder respetarla en futuras visitas. Y al almacenar la preferencia, genera un evento que permite a otras ventanas que muestren el mismo sitio cumplir también con la solicitud.

Como otro ejemplo, imagine una aplicación de edición de imágenes basada en la web que permite al usuario mostrar paletas de herramientas en ventanas separadas. Cuando el usuario selecciona una herramienta, la aplicación utiliza localStorage para guardar el estado actual y generar una notificación a otras ventanas de que se ha seleccionado una nueva herramienta.

### **15.12.2 Cookies**

Una *cookie* es una pequeña cantidad de datos con nombre almacenada por el navegador web y asociada a una página web o sitio web concreto. Las cookies fueron diseñadas para la programación del lado del servidor, y en el nivel más bajo, se

implementan como una extensión del protocolo HTTP. Los datos de las cookies se transmiten automáticamente entre el navegador web y el servidor web, por lo que los scripts del lado del servidor pueden leer y escribir los valores de las cookies que se almacenan en el cliente. Esta sección demuestra cómo los scripts del lado del cliente también pueden manipular las cookies utilizando la propiedad cookie del objeto Document.

#### ¿POR QUÉ "COOKIE"?

El nombre "cookie" no tiene mucha importancia, pero no se utiliza sin precedentes. En los anales de la historia de la informática, el término "cookie" o "cookie mágica" se ha utilizado para referirse a un pequeño trozo de datos, en particular un trozo de datos privilegiados o secretos, similares a una contraseña, que demuestra la identidad o permite el acceso. En JavaScript, las cookies se utilizan para guardar el estado y pueden establecer una especie de identidad para un navegador web. Sin embargo, las cookies en JavaScript no utilizan ningún tipo de criptografía y no son seguras de ninguna manera (aunque transmitirlas a través de una conexión https: ayuda).

La API para manipular las cookies es antigua y críptica. No hay métodos implicados: las cookies se consultan, se establecen y se eliminan leyendo y escribiendo la propiedad cookie del objeto Document utilizando cadenas con un formato especial. El tiempo de vida y el alcance de cada cookie pueden ser especificados individualmente con atributos de cookie. Estos atributos también se especifican con cadenas especialmente formateadas establecidas en la misma propiedad cookie.

Las subsecciones siguientes explican cómo consultar y establecer los valores y atributos de las cookies.

#### GALLETAS DE LECTURA

Cuando se lee la propiedad document.cookie, ésta devuelve una cadena que contiene todas las cookies que se aplican al

documento actual. La cadena es una lista de pares nombre/valor separados entre sí por un punto y coma y un espacio. El valor de la cookie es sólo el valor en sí mismo y no incluye ninguno de los atributos que puedan estar asociados a esa cookie. (Hablaremos de los atributos a continuación.) Para poder hacer uso de la propiedad `document.cookie`, normalmente hay que llamar al método `split()` para dividirla en pares nombre/valor individuales.

Una vez que haya extraído el valor de una cookie de la propiedad de la cookie, debe interpretar ese valor basándose en cualquier formato o codificación que haya utilizado el creador de la cookie. Por ejemplo, puede pasar el valor de la cookie a `decodeURIComponent()` y luego a `JSON.parse()`.

El código que sigue define una función `getCookie()` que analiza la propiedad `document.cookie` y devuelve un objeto cuyas propiedades especifican los nombres y valores de las cookies del documento:

```
// Devuelve las cookies del documento como un objeto Map.  
// Asumir que los valores de las cookies están codificados con encodeURIComponent().  
function getCookies() { let cookies = new Map(); // El objeto que devolveremos let all =  
document.cookie; // Obtener todas las cookies en una gran cadena let list = all;  
split("; "); // Dividir en pares individuales de nombre/valor for(let cookie of list) { //  
Para cada cookie en esa lista if (! cookie.includes("=")) continue; // Omitir si no hay  
signo = let p = cookie.indexOf("="); // Encontrar el primer signo = let name = cookie.  
substring(0, p); // Obtener el nombre de la cookie let value = cookie.substring(p+1); //  
Obtener el valor de la cookie value = decodeURIComponent(value); // Decodificar el  
valor cookies. set(name, value); // Recordar el nombre y el valor de la cookie } return  
cookies; }
```

## ATRIBUTOS DE LAS COOKIES: VIDA ÚTIL Y ÁMBITO DE APLICACIÓN

Además de un nombre y un valor, cada cookie tiene atributos opcionales que controlan su duración y alcance. Antes de describir cómo configurar las cookies con JavaScript, debemos explicar los atributos de las cookies.

Las cookies son transitorias por defecto; los valores que almacenan duran mientras dura la sesión del navegador web, pero se pierden cuando el usuario sale del navegador. Si quieres que una cookie dure más allá de una sola sesión de navegación, debes indicarle al navegador cuánto tiempo (en segundos) quieras que conserve la cookie especificando un atributo de edad máxima. Si especifica un tiempo de vida, el navegador almacenará las cookies en un archivo y las borrará sólo cuando expiren.

La visibilidad de las cookies tiene un alcance por el origen del documento, como lo tienen localStorage y sessionStorage, pero también por la ruta del documento. Este ámbito es configurable a través de los atributos de la cookie path y domain. Por defecto, una cookie está asociada y es accesible a la página web que la creó y a cualquier otra página web en el mismo directorio o en cualquier subdirectorío de ese directorio. Si la página web *example.com/catalog/index.html* crea una cookie, por ejemplo, esa cookie también es visible para *example.com/catalog/order.html* y *example.com/catalog/widgets/index.html*, pero no es visible para *example.com/about.html*.

Este comportamiento de visibilidad por defecto es a menudo exactamente lo que usted quiere. Sin embargo, a veces querrá

utilizar los valores de las cookies en todo el sitio web, independientemente de la página que cree la cookie. Por ejemplo, si el usuario introduce su dirección postal en un formulario de una página, es posible que desee guardar esa dirección para utilizarla como valor predeterminado la próxima vez que vuelva a la página y también como valor predeterminado en un formulario no relacionado en otra página donde se le pida que introduzca una dirección de facturación. Para permitir este uso, se especifica una ruta para la cookie. Entonces, cualquier página web del mismo servidor web cuya URL comience con el prefijo de la ruta que usted especificó puede compartir la cookie. Por ejemplo, si una cookie establecida por *example.com/catalog/widgets/index.html* tiene su ruta establecida en "/catalog", esa cookie también es visible para *example.com/catalog/order.html*. O, si la ruta se establece como "/", la cookie es visible para cualquier página en el dominio *example.com*, dando a la cookie un alcance como el de `localStorage`.

Por defecto, las cookies son delimitadas por el origen del documento. Sin embargo, los sitios web grandes pueden querer que las cookies se compartan entre subdominios. Por ejemplo, el servidor de *order.example.com* puede necesitar leer los valores de las cookies establecidas en *catalog.example.com*. Aquí es donde entra en juego el atributo de dominio. Si una cookie creada por una página en *catalog.example.com* establece su atributo de ruta a "/" y su atributo de dominio a ".example.com", esa cookie está disponible para todas las páginas web en *catalog.example.com*, *orders.example.com* y cualquier otro servidor en el dominio

*example.com*. Tenga en cuenta que no puede establecer el dominio de una cookie a un dominio que no sea el dominio principal de su servidor.

El último atributo de las cookies es un atributo booleano llamado secure que especifica cómo se transmiten los valores de las cookies a través de la red. Por defecto, las cookies son inseguras, lo que significa que se transmiten a través de una conexión HTTP normal e insegura. Sin embargo, si una cookie está marcada como segura, sólo se transmite cuando el navegador y el servidor están conectados a través de HTTPS u otro protocolo seguro.

#### LIMITACIONES DE LAS COOKIES

Las cookies están pensadas para almacenar pequeñas cantidades de datos mediante scripts del lado del servidor, y esos datos se transfieren al servidor cada vez que se solicita una URL relevante. La norma que define las cookies anima a los fabricantes de navegadores a permitir un número ilimitado de cookies de tamaño ilimitado, pero no exige que los navegadores retengan más de 300 cookies en total, 20 cookies por servidor web o 4 KB de datos por cookie (tanto el nombre como el valor cuentan para este límite de 4 KB). En la práctica, los navegadores permiten muchas más de 300 cookies en total, pero es posible que algunos sigan aplicando el límite de 4 KB de tamaño.

## ALMACENAMIENTO DE COOKIES

Para asociar un valor transitorio de la cookie con el documento actual, simplemente establezca la propiedad de la cookie a una cadena name=value. Por ejemplo:

```
document.cookie = `version=${encodeURIComponent(document.lastModified)}`;
```

La próxima vez que lea la propiedad cookie, el par nombre/valor almacenado se incluirá en la lista de cookies del documento. Los valores de las cookies no pueden incluir puntos y comas, ni espacios en blanco. Por este motivo, puede utilizar la función

global del núcleo de JavaScript encodeURIComponent() para codificar el valor antes de almacenarlo en la cookie. Si hace esto, tendrá que utilizar la función correspondiente decodeURIComponent() cuando lea el valor de la cookie.

Una cookie escrita con un simple par nombre/valor dura la sesión actual de navegación web pero se pierde cuando el usuario sale del navegador. Para crear una cookie que pueda durar más allá de las sesiones del navegador, especifique su duración (en segundos) con un atributo max-age. Puede hacerlo estableciendo la propiedad de la cookie a una cadena de la forma: name=value; max-age=seconds. La siguiente función establece una cookie con un atributo opcional de edad máxima:

```
// Almacenar el par nombre/valor como una cookie, codificando el valor con //  
// encodeURIComponent() para escapar de puntos y comas y espacios.  
// Si daysToLive es un número, establece el atributo max-age para que la cookie //  
// expire después del número de días especificado. Pase 0 para eliminar una cookie.  
function setCookie(name, value, daysToLive=null) { let cookie =  
` ${name}=${encodeURIComponent(value)} `; if (daysToLive !== null) { cookie += `;  
max-age=${daysToLive*60*24}`; } document.cookie = cookie;  
}  
}
```

Del mismo modo, puede establecer los atributos de ruta y dominio de una cookie añadiendo cadenas de la forma ;ruta=valor o ;dominio=valor a la cadena que estableció en la propiedad document.cookie. Para establecer la propiedad secure, simplemente añada ;secure.

Para cambiar el valor de una cookie, vuelva a establecer su valor utilizando el mismo nombre, ruta y dominio junto con el nuevo valor. Puede cambiar el tiempo de vida de una cookie cuando cambie su valor especificando un nuevo atributo max-age.

Para eliminar una cookie, configúrela de nuevo utilizando el mismo nombre, ruta y dominio, especificando un valor arbitrario (o vacío) y un atributo de edad máxima de 0.

### **15.12.3 IndexedDB**

La arquitectura de las aplicaciones web ha contado tradicionalmente con HTML, CSS y JavaScript en el cliente y una base de datos en el servidor. Por lo tanto, puede que te sorprenda saber que la plataforma web incluye una simple base de datos de objetos con una API de JavaScript para almacenar de forma persistente objetos de JavaScript en el ordenador del usuario y recuperarlos cuando sea necesario.

IndexedDB es una base de datos de objetos, no una base de datos relacional, y es mucho más simple que las bases de datos que soportan consultas SQL. Sin embargo, es más potente, eficiente y robusta que el almacenamiento clave/valor proporcionado por el localStorage. Al igual que el localStorage, las bases de datos IndexedDB se ajustan al origen del documento que las contiene: dos páginas web con el mismo origen pueden acceder a los datos de la otra, pero las páginas web de diferentes orígenes no pueden hacerlo.

Cada origen puede tener cualquier número de bases de datos IndexedDB. Cada una tiene un nombre que debe ser único dentro

del origen. En la API de IndexedDB, una base de datos es simplemente una colección de almacenes de *objetos* con nombre. Como su nombre indica, un almacén de objetos almacena objetos. Los objetos se serializan en el almacén de objetos utilizando el algoritmo de clonación estructurado (véase "[El algoritmo de clonación estructurado](#)"), lo que significa que los objetos que se almacenan pueden tener propiedades cuyos valores son Mapas, Conjuntos o matrices tipadas. Cada objeto debe tener una *clave* por la que pueda ordenarse y recuperarse del almacén. Las claves deben ser únicas -dos objetos del mismo almacén no pueden tener la misma clave- y deben tener un orden natural para que puedan ser ordenadas. Las cadenas de JavaScript, los números y los objetos Date son claves válidas. Una base de datos IndexedDB puede generar automáticamente una clave única para cada objeto que se inserte en la base de datos. Sin embargo, a menudo los objetos que se insertan en un almacén de objetos ya tienen una propiedad que puede utilizarse como clave. En este caso, se especifica una "ruta de clave" para esa propiedad cuando se crea el almacén de objetos. Conceptualmente, una ruta de clave es un valor que indica a la base de datos cómo extraer la clave de un objeto.

Además de recuperar objetos de un almacén de objetos por su valor de clave primaria, es posible que desee poder buscar basándose en el valor de otras propiedades del objeto. Para poder hacer esto, puedes definir cualquier número de *índices* en el almacén de objetos. (La capacidad de indexar un almacén de objetos explica el nombre "IndexedDB".) Cada índice define una clave secundaria para los objetos almacenados. Estos índices no

suelen ser únicos, y varios objetos pueden coincidir con un mismo valor de clave.

IndexedDB ofrece garantías de atomicidad: las consultas y actualizaciones de la base de datos se agrupan dentro de una *transacción*, de modo que todas ellas tienen éxito o fallan juntas y nunca dejan la base de datos en un estado indefinido y parcialmente actualizado. Las transacciones en IndexedDB son más sencillas que en muchas API de bases de datos; las mencionaremos de nuevo más adelante.

Conceptualmente, la API de IndexedDB es bastante sencilla. Para consultar o actualizar una base de datos, primero se abre la base de datos que se desea (especificándola por su nombre). A continuación, se crea un objeto de transacción y se utiliza ese objeto para buscar el almacén de objetos deseado dentro de la base de datos, también por su nombre. Finalmente, se busca un objeto llamando al método `get()` del almacén de objetos o se almacena un nuevo objeto llamando a `put()` (o llamando a `add()`, si se quiere evitar sobrescribir los objetos existentes).

Si quieres buscar los objetos de un rango de claves, crea un objeto `IDBRange` que especifique los límites superior e inferior del rango y pásalo a los métodos `getAll()` o `openCursor()` del almacén de objetos.

Si quieres hacer una consulta utilizando una clave secundaria, busca el índice con nombre del almacén de objetos, y luego llama a los métodos `get()`, `getAll()`, o `openCursor()` del objeto índice, pasando una sola clave o un objeto `IDBRange`.

Esta simplicidad conceptual de la API de IndexedDB se complica, sin embargo, por el hecho de que la API es asíncrona (para que las aplicaciones web puedan utilizarla sin bloquear el hilo principal de la interfaz de usuario del navegador). IndexedDB se definió antes de que las promesas fueran ampliamente soportadas, por lo que la API está basada en eventos en lugar de en promesas, lo que significa que no funciona con `async` y `await`.

La creación de transacciones y la búsqueda de almacenes de objetos e índices son operaciones sincrónicas. Pero abrir una base de datos, actualizar un almacén de objetos y consultar un almacén o índice son operaciones asíncronas. Todos estos métodos asíncronos devuelven inmediatamente un objeto de petición. El navegador desencadena un evento de éxito o error en el objeto `request` cuando la petición tiene éxito o falla, y puedes definir manejadores con las propiedades `onsuccess` y `onerror`. Dentro de un manejador `onsuccess`, el resultado de la operación está disponible como la propiedad `result` del objeto `request`. Otro evento útil es el evento "complete" que se envía a los objetos de transacción cuando una transacción se ha completado con éxito.

Una característica conveniente de esta API asíncrona es que simplifica la gestión de las transacciones. La API de IndexedDB le obliga a crear un objeto de transacción para obtener el almacén de objetos sobre el que puede realizar consultas y actualizaciones. En una API sincrónica, se esperaría marcar explícitamente el final de la transacción llamando a un método `commit()`. Pero con IndexedDB, las transacciones se consignan automáticamente (si no las aborta explícitamente) cuando se han

ejecutado todos los manejadores de eventos onsuccess y no hay más peticiones asíncronas pendientes que hagan referencia a esa transacción.

Hay un evento más que es importante para la API de IndexedDB. Cuando se abre una base de datos por primera vez, o cuando se incrementa el número de versión de una base de datos existente, IndexedDB lanza un evento "upgradeneed" en el objeto de petición devuelto por la llamada indexedDB.open(). El trabajo del manejador de eventos "upgradeneed" es definir o actualizar el esquema para la nueva base de datos (o la nueva versión de la base de datos existente). Para las bases de datos IndexedDB, esto significa crear almacenes de objetos y definir índices en esos almacenes de objetos. Y de hecho, la única vez que la API de IndexedDB permite crear un almacén de objetos o un índice es en respuesta a un evento "upgradeneed".

Con esta visión general de IndexedDB en mente, ahora debería ser capaz de entender el [Ejemplo 15-13](#). Este ejemplo utiliza IndexedDB para crear y consultar una base de datos que mapea los códigos postales de EE.UU. a Ciudades de Estados Unidos. Demuestra muchas, pero no todas, las características básicas de IndexedDB. [El ejemplo 15-13](#) es largo, pero está bien comentado.

*Ejemplo 15-13. Una base de datos IndexedDB de códigos postales de Estados Unidos*

---

```
// Esta función de utilidad obtiene asíncoricamente el objeto de la base de datos
// (creando // e inicializando la BD si es necesario) y lo pasa a la llamada de retorno.
function withDB(callback) { let request = indexedDB.open("zipcodes", 1); // Solicitar
v1 de la base de datos
request.onerror = console.error; // Registrar cualquier error
```

`request.onsuccess = () => { // O llamar a esto cuando haya terminado let db = request.result; // El resultado de la solicitud es la base de datos callback(db); // Invocar el callback con la base de datos };`

`// Si la versión 1 de la base de datos aún no existe, entonces este evento // se activará el manejador. Esto se utiliza para crear y`

```
inicializar
  // almacenes de objetos e índices cuando se crea la BD por primera vez o para modificarlos
  // cuando pasamos de una versión del esquema de la BD a otra.
  request.onupgradeneed = () => { initdb(request.result, callback); };

// withDB() llama a esta función si la base de datos no ha sido inicializada todavía.
// Configuramos la base de datos y la llenamos de datos, luego pasamos la base de datos a
// la función de devolución de llamada.
//
// Nuestra base de datos de códigos postales incluye un almacén de objetos que contiene
// objetos como este:
//
// {
// código postal: "02134",
// ciudad: "Allston",
// estado: "MA",
// }
//
// Utilizamos la propiedad "zipcode" como clave de la base de datos y creamos un índice
// para el nombre de la ciudad. function initdb(db, callback) {
  // Crear el almacén de objetos, especificando un nombre para el almacén y
  // un objeto de opciones que incluye la "ruta de la clave" especificando el
  // nombre de la propiedad del campo clave para este almacén.
  let store = db.createObjectStore("zipcodes", // nombre de la tienda { keyPath: "zipcode"
});

// Ahora indexa el almacén de objetos por el nombre de la ciudad así como por el código
// postal.
  // Con este método la cadena de la ruta de acceso a la clave se pasa directamente como
  // argumento requerido en lugar de como parte de un objeto de opciones. store.
createIndex("cities", "city");

// Ahora obtenemos los datos con los que vamos a inicializar la base de datos.
```

```
// El archivo de datos zipcodes.json se generó a partir de los datos de CClicensed de
//      www.geonames.org:      https://download.geonames.org/export/zip/US.zip
fetch("zipcodes.json") // Realiza una petición HTTP GET . then(response => response.json())
// Analizar el cuerpo como JSON . then(zipcodes => { // Obtener 40K registros de códigos
postales
    // Para insertar los datos del código postal en la base de datos necesitamos un
    // objeto de transacción. Para crear nuestro objeto de transacción, necesitamos
    // para especificar qué almacenes de objetos vamos a utilizar
(sólo tenemos
    // uno) y tenemos que decirle que vamos a hacer escrituras en la // base de datos,
no sólo lecturas:
    let transaction = db.transaction(["zipcodes"], "readwrite"); transaction.
onerror = console.error;

    // Obtener nuestro almacén de objetos de la transacción let store = transaction.
objectStore("zipcodes");

    // Lo mejor de la API de IndexedDB es que los almacenes de objetos // son
*realmente* sencillos. Así es como añadimos (o actualizamos) nuestros registros: for(let
record of zipcodes) { store.put(record); }

    // Cuando la transacción se completa con éxito, la base de datos
    // está inicializado y listo para su uso, por lo que podemos llamar al
    // función de devolución de llamada que se pasó originalmente a withDB()
    transaction.oncomplete = () => { callback(db); }; });

}

// Dado un código postal, utiliza la API de IndexedDB para buscar asíncronamente la
ciudad
// con ese código postal, y pasarlo a la llamada de retorno especificada,
```

*o pasar null si // no se encuentra ninguna ciudad.*

```
function lookupCity(zip, callback) { withDB(db => {
    // Crear un objeto de transacción de sólo lectura para esta consulta. El // argumento
    // es un array de almacenes de objetos que necesitaremos utilizar. let transaction = db.
    transaction(["zipcodes"]);

    // Obtener el almacén de objetos de la transacción let zipcodes = transaction.
    objectStore("zipcodes");

    // Ahora solicita el objeto que coincide con la clave de código postal especificada.
    // Las líneas anteriores eran sincrónicas, pero esta es asíncrona.

    let request = zipcodes.get(zip); request.onerror = console.error; // Registrar los
    errores request.onsuccess = () => { // O llamar a esta función en caso de éxito let record
    = request.result; // Este es el resultado de la consulta if (record) { // Si encontramos una
    coincidencia, pásala al callback callback(`${record.city}, ${record.state}`);
    } else { // En caso contrario, indicar al callback que hemos fallado
        callback(null);
    };
});
}

// Dado el nombre de una ciudad, utiliza la API de IndexedDB para asíncrono
// buscar todos los registros de códigos postales de todas las ciudades (en cualquier
// estado) que tengan // ese nombre (que distingue entre mayúsculas y minúsculas).
function lookupZipcodes(city, callback) { withDB(db => {
    // Como en el caso anterior, creamos una transacción y obtenemos el almacén de
    objetos
```

```

let transaction = db.transaction(["zipcodes"]); let store = transaction.
objectStore("zipcodes");

// Esta vez también obtenemos el índice de ciudades del objeto store let index = store.
index("cities");

// Pedir todos los registros coincidentes en el índice con el nombre de la ciudad
// especificada, y cuando los obtenemos los pasamos a la llamada de retorno. // Si
// esperáramos más resultados, podríamos usar openCursor() en su lugar.

let request = index.getAll(city); request.onerror = console.error;
request.onsuccess = () => { callback(request.result); };

});

}

```

## 15.13 Hilos de trabajo y mensajería

Una de las características fundamentales de JavaScript es que es singlethreaded: un navegador nunca ejecutará dos manejadores de eventos al mismo tiempo, y nunca activará un temporizador mientras se esté ejecutando un manejador de eventos, por ejemplo. Las actualizaciones concurrentes del estado de la aplicación o del documento simplemente no son posibles, y los programadores del lado del cliente no necesitan pensar, o incluso entender, la programación concurrente. Un corolario es que las funciones JavaScript del lado del cliente no deben ejecutarse demasiado tiempo; de lo contrario, atarán el bucle de eventos y el navegador web dejará de responder a la entrada del usuario. Esta es la razón por la que `fetch()` es una función asíncrona, por ejemplo.

Los navegadores web relajan cuidadosamente el requisito de un solo hilo con la clase `Worker`: las instancias de esta clase

representan hilos que se ejecutan simultáneamente con el hilo principal y el bucle de eventos. Los trabajadores viven en un entorno de ejecución autónomo con un objeto global completamente independiente y sin acceso a los objetos Window o Document. Los trabajadores pueden comunicarse con el hilo principal sólo a través del paso de mensajes asíncronos. Esto significa que las modificaciones concurrentes del DOM siguen siendo imposibles, pero también significa que puedes escribir funciones de larga duración que no paralicen el bucle de eventos y cuelguen el navegador. La creación de un nuevo worker no es una operación pesada como la de abrir una nueva ventana del navegador, pero los workers tampoco son "fibras" ligeras, y no tiene sentido crear nuevos workers para realizar operaciones triviales. Las aplicaciones web complejas pueden encontrar útil la creación de decenas de workers, pero es poco probable que una aplicación con cientos o miles de workers sea práctica.

Los workers son útiles cuando tu aplicación necesita realizar tareas de cálculo intensivo, como el procesamiento de imágenes. El uso de un worker desplaza este tipo de tareas fuera del hilo principal para que el navegador no deje de responder. Y los workers también ofrecen la posibilidad de dividir el trabajo entre múltiples hilos. Pero los workers también son útiles cuando tienes que realizar frecuentes cálculos moderadamente intensivos. Supongamos, por ejemplo, que estás implementando un simple editor de código en el navegador, y quieres incluir el resaltado de sintaxis. Para que el resultado sea correcto, tienes que analizar el código en cada pulsación de tecla. Pero si lo haces en el hilo principal, es probable que el código de análisis impida que los manejadores de eventos que responden a las pulsaciones

de las teclas del usuario se ejecuten rápidamente y la experiencia de escritura del usuario sea lenta.

Como con cualquier API de hilos, hay dos partes en la API del trabajador. La primera es el objeto `Worker`: este es el aspecto de un worker desde el exterior, para el hilo que lo crea. La segunda es el objeto

`WorkerGlobalScope`: es el objeto global para un nuevo worker, y es lo que parece un hilo worker, por dentro, para sí mismo.

Las siguientes secciones cubren `Worker` y `WorkerGlobalScope` y también explican la API de paso de mensajes que permite a los trabajadores comunicarse con el hilo principal y entre sí. La misma API de comunicación se utiliza para intercambiar mensajes entre un documento y los elementos `<iframe>` contenidos en el documento, y esto se cubre también en las siguientes secciones.

### 15.13.1 Objetos de trabajo

Para crear un nuevo trabajador, llame al constructor `Worker()`, pasando una URL que especifica el código JavaScript que el trabajador debe ejecutar:

```
let dataCruncher = new Worker("utils/cruncher.js");
```

Si especifica una URL relativa, se resuelve en relación con la URL del documento que contiene el script que llamó al constructor `Worker()`. Si especifica una URL absoluta, debe tener el mismo origen (mismo protocolo, host y puerto) que el documento que lo contiene.

Una vez que tengas un objeto Worker, puedes enviarle datos con `postMessage()`. El valor que pases a `postMessage()` se copiará utilizando el algoritmo de clonación estructurado (ver "[El algoritmo de clonación estructurado](#)"), y la copia resultante se entregará al trabajador a través de un evento de mensaje:



```
dataCruncher.postMessage("/api/data/to/crunch");
```

Aquí sólo estamos pasando un único mensaje de cadena, pero también puedes utilizar objetos, arrays, arrays tipados, Mapas, Sets, etc. Puedes recibir mensajes de un trabajador escuchando los eventos "mensaje" en el objeto Worker:

```
dataCruncher.onmessage = function(e) { let stats = e.data; // El mensaje es la  
propiedad data del evento console.log(`Promedio: ${stats.mean}`); }
```

Como todos los objetivos de eventos, los objetos Worker definen el estándar addEventListener() y removeEventListener(), y puedes utilizarlos en lugar de onmessage.

Además de postMessage(), los objetos Worker sólo tienen otro método, terminate(), que obliga a un hilo worker a dejar de ejecutarse.

### 15.13.2 El objeto global en los trabajadores

Cuando se crea un nuevo worker con el constructor Worker(), se especifica la URL de un archivo de código JavaScript. Ese código se ejecuta en un nuevo y prístino entorno de ejecución de JavaScript, aislado del script que creó el trabajador. El objeto global para ese nuevo entorno de ejecución es un objeto WorkerGlobalScope. Un WorkerGlobalScope es algo más que el objeto global del núcleo de JavaScript, pero menos que un objeto Window del lado del cliente.

El objeto WorkerGlobalScope tiene un método postMessage() y una propiedad manejadora de eventos onmessage que son iguales a los del objeto Worker pero funcionan en la dirección opuesta: llamar a postMessage() dentro de un worker genera un evento de mensaje fuera del worker, y los mensajes enviados desde fuera del worker son convertidos en eventos y entregados al manejador onmessage. Dado que el WorkerGlobalScope es el objeto global de un worker, postMessage() y onmessage parecen una función global y una variable global para el código del worker.

Si se pasa un objeto como segundo argumento al constructor Worker(), y si ese objeto tiene una propiedad name, entonces el valor de esa propiedad se convierte en el valor de la propiedad name en el objeto global del worker. Un trabajador puede incluir este nombre en cualquier mensaje que imprima con console.warn() o console.error().

La función close() permite que un trabajador se termine a sí mismo, y es similar en efecto al método terminate() de un objeto Worker.

Dado que WorkerGlobalScope es el objeto global para los trabajadores, tiene todas las propiedades del objeto global del núcleo de JavaScript, como el objeto JSON, la función isNaN() y el constructor Date(). Además, sin embargo, WorkerGlobalScope también tiene las siguientes propiedades del objeto Window del lado del cliente:

- self es una referencia al propio objeto global.

WorkerGlobalScope no es un objeto Window y no define una propiedad de ventana.

- Los métodos del temporizador setTimeout(), clearTimeout(), setInterval() y clearInterval().
- Una propiedad de ubicación que describe la URL que se pasó al constructor de Worker(). Esta propiedad se refiere a un objeto Location, al igual que la propiedad location de una ventana. El objeto Location tiene las propiedades href, protocol, host, hostname, port, pathname, search y hash.  
Sin embargo, en un trabajador, estas propiedades son de sólo lectura.
- Una propiedad del navegador que se refiere a un objeto con propiedades como las del objeto Navigator de una ventana. El objeto Navigator de un trabajador tiene las propiedades appName, appVersion, platform, userAgent y onLine.
- Los métodos habituales de destino de eventos addEventListener() y removeEventListener().

Por último, el objeto WorkerGlobalScope incluye importantes APIs de JavaScript del lado del cliente, como el objeto Console, la función fetch() y la API IndexedDB. WorkerGlobalScope también incluye el constructor Worker(), lo que significa que los hilos de trabajo pueden crear sus propios trabajadores.

### 15.13.3 Importar código a un trabajador

Los workers se definieron en los navegadores web antes de que JavaScript tuviera un sistema de módulos, por lo que los workers tienen un sistema único para incluir código adicional.

WorkerGlobalScope define importScripts() como una función global a la que todos los workers tienen acceso:

```
// Antes de empezar a trabajar, carga las clases y utilidades que necesitaremos
importScripts("utils/Histogram.js", "utils/BitSet.js");
```

importScripts() toma uno o más argumentos de URL, cada uno de los cuales debe referirse a un archivo de código JavaScript. Las URLs relativas se resuelven en relación con la URL que se pasó al constructor de Worker() (no en relación con el documento contenedor).

importScripts() carga y ejecuta sincrónicamente estos archivos uno tras otro, en el orden en que fueron especificados. Si la carga de un script provoca un error de red, o si la ejecución arroja un error de cualquier tipo, no se carga ni se ejecuta ninguno de los scripts posteriores. Un script cargado con importScripts() puede llamar a su vez a

importScripts() para cargar los archivos de los que depende. Tenga en cuenta, sin embargo, que importScripts() no intenta hacer un seguimiento de los scripts que ya se han cargado y no hace nada para evitar los ciclos de dependencia.

importScripts() es una función sincrónica: no devuelve hasta que todos los scripts se hayan cargado y ejecutado. Puede empezar a utilizar los scripts cargados tan pronto como vuelva importScripts(): no es necesario un callback, un manejador de eventos, un método then() o un await.

Una vez que se ha interiorizado la naturaleza asíncrona de JavaScript del lado del cliente, resulta extraño volver a la programación simple y sincrónica. Pero esa es la belleza de los hilos: puedes utilizar una llamada a una función de bloqueo en un trabajador sin bloquear el bucle de eventos en el hilo principal, y sin bloquear los cálculos que se realizan simultáneamente en otros trabajadores.

## MÓDULOS EN LOS TRABAJADORES

Para utilizar módulos en los trabajadores, debes pasar un segundo argumento al constructor de `Worker()`. Este segundo argumento debe ser un objeto con una propiedad `type` establecida a la cadena "module". Pasar la opción `type: "module"` al constructor de `Worker()` es muy parecido a utilizar el atributo `type="module"` en una etiqueta HTML `<script>`: significa que el código debe ser interpretado como un módulo y

que importaciones permiten las declaraciones.

Cuando un trabajador carga un módulo en lugar de un script tradicional, el `WorkerGlobalScope` no define la función `importScripts()`.

Tenga en cuenta que a principios de 2020, Chrome es el único navegador que admite verdaderos módulos y declaraciones en los

### 15.13.4 Modelo de ejecución de los trabajadores

Los hilos de trabajo ejecutan su código (y todos los scripts o módulos importados) de forma sincrónica de arriba a abajo, y luego entran en una fase asíncrona en la que responden a eventos y temporizadores. Si un worker registra un manejador de eventos de "mensajes", nunca saldrá mientras exista la posibilidad de que sigan llegando eventos de mensajes. Pero si un trabajador no escucha mensajes, se ejecutará hasta que no haya más tareas pendientes (como promesas `fetch()` y temporizadores) y se hayan llamado todas las llamadas de retorno relacionadas con la tarea. Una vez que todos los callbacks registrados han sido llamados, no hay manera de que un trabajador pueda comenzar una nueva tarea, por lo que es seguro que el hilo salga, lo que hará automáticamente. Un trabajador también puede cerrarse explícitamente llamando a la función global `close()`. Ten en cuenta que no hay propiedades o métodos en el objeto `Worker` que especifiquen si un hilo worker sigue ejecutándose o no, por lo que los workers no deben cerrarse a sí mismos sin coordinarlo de alguna manera con su hilo padre.

## ERRORES EN LOS TRABAJADORES

Si se produce una excepción en un trabajador y no es capturada por ninguna cláusula catch, se lanza un evento "error" en el objeto global del trabajador. Si este evento es manejado y el manejador llama al método preventDefault() del objeto del evento, la propagación del error termina. En caso contrario, el evento "error" se dispara en el objeto Worker. Si se llama a preventDefault() allí, la propagación termina. En caso contrario, se imprime un mensaje de error en la consola del desarrollador y se invoca el manejador onerror ([§15.1.7](#)) del objeto Window.

```
// Maneja los errores del trabajador no capturado con un manejador dentro del trabajador.  
self.onerror = function(e) { console.log(`Error in worker at ${e.filename}:${e.lineno}: ${e.message}`);  
    e.preventDefault(); };  
  
// O bien, manejar los errores del trabajador no capturados con un manejador fuera del trabajador.  
worker.onerror = function(e) { console.log(`Error in worker at ${e.filename}:${e.lineno}: ${e.message}`);  
    e.preventDefault(); };
```

Al igual que las ventanas, los trabajadores pueden registrar un manejador para ser invocado cuando una promesa es rechazada y no hay una función .catch() para manejarla. Dentro de un trabajador se puede detectar esto definiendo una función self.onunhandledrejection o utilizando addEventListener() para registrar un manejador global para eventos "unhandledrejection". El objeto de evento pasado a este manejador tendrá una propiedad promise cuyo valor es el objeto

Promise que rechazó y una propiedad reason cuyo valor es lo que se habría pasado a una función .catch().

### 15.13.5 postMessage(), MessagePorts y Canales de mensajes

El método postMessage() del objeto Worker y la función global postMesage() definida dentro de un worker funcionan invocando los métodos postMessage() de un par de objetos MessagePort que se crean automáticamente junto con el worker. El JavaScript del lado del cliente no puede acceder directamente a estos objetos MessagePort creados automáticamente, pero puede crear nuevos pares de puertos conectados con el constructor MessageChannel():

```
let channel = new MessageChannel; // Crea un nuevo canal. let myPort = channel.  
port1; // Tiene dos puertos let yourPort = channel.port2; // conectados entre sí.  
  
myPort.postMessage("¿Puedes oírmeme?"); // Un mensaje enviado a uno será  
yourPort.onmessage = (e) => console.log(e.data); // ser recibido en el otro.
```

Un MessageChannel es un objeto con las propiedades port1 y port2 que se refieren a un par de objetos MessagePort conectados. Un MessagePort es un objeto con un método postMessage() y una propiedad manejadora de eventos onmessage. Cuando se llama a postMessage() en un puerto de un par conectado, se dispara un evento "mensaje" en el otro puerto del par. Puedes recibir estos eventos "mensaje" estableciendo la propiedad onmessage o utilizando addEventListener() para registrar un oyente de eventos "mensaje".

Los mensajes enviados a un puerto se ponen en cola hasta que se define la propiedad onmessage o hasta que se llama al método

`start()` en el puerto. Esto evita que los mensajes enviados por un extremo del canal sean perdidos por el otro extremo. Si utiliza `addEventListener()` con un `MessagePort`, no olvide llamar a `start()` o puede que nunca vea un mensaje entregado.

Todas las llamadas a `postMessage()` que hemos visto hasta ahora han tomado un único argumento de mensaje. Pero el método también acepta un segundo argumento opcional. Este segundo argumento es un array de elementos que deben ser transferidos al otro extremo del canal en lugar de tener una copia enviada a través del canal. Los valores que pueden ser transferidos en lugar de copiados son `MessagePorts` y `ArrayBuffers`. (Algunos navegadores también implementan otros tipos transferibles, como `ImageBitmap` y `OffscreenCanvas`. Sin embargo, estos no son universalmente soportados y no se cubren en este libro). Si el primer argumento de `postMessage()` incluye un `MessagePort` (anidado en cualquier lugar dentro del objeto mensaje), entonces ese `MessagePort` debe aparecer también en el segundo argumento. Si haces esto, entonces el `MessagePort` estará disponible para el otro extremo del canal e inmediatamente dejará de ser funcional en tu extremo.

Supongamos que has creado un trabajador y quieres tener dos canales para comunicarte con él: un canal para el intercambio ordinario de datos y otro para los mensajes de alta prioridad. En el hilo principal, podrías crear un `MessageChannel`, y luego llamar a `postMessage()` en el trabajador para pasárselo uno de los `MessagePorts`:

```
let worker = new Worker("worker.js"); let urgentChannel = new  
MessageChannel(); let urgentPort = urgentChannel.port1; worker.  
postMessage({ command: "setUrgentPort", value:  
urgentChannel.port2 },  
[ urgentChannel.port2 ]);  
// Ahora podemos recibir mensajes urgentes del trabajador así
```

```
urgentPort.addEventListener("message", handleUrgentMessage); urgentPort.  
start(); // Empezar a recibir mensajes // Y enviar mensajes urgentes así urgentPort.  
postMessage("test");
```

Los MessageChannels también son útiles si creas dos workers y quieres que se comuniquen directamente entre ellos en lugar de requerir que el código del hilo principal transmita los mensajes entre ellos.

El otro uso del segundo argumento de postMessage() es transferir ArrayBuffers entre trabajadores sin tener que copiarlos. Esta es una importante mejora de rendimiento para grandes ArrayBuffers como los que se utilizan para mantener los datos de las imágenes. Cuando un ArrayBuffer se transfiere a través de un MessagePort, el ArrayBuffer se vuelve inutilizable en el hilo original, por lo que no hay posibilidad de acceso concurrente a su contenido. Si el primer argumento de postMessage() incluye un ArrayBuffer, o cualquier valor (como un array tipificado) que tenga un ArrayBuffer, entonces ese buffer puede aparecer como un elemento del array en el segundo argumento de postMessage(). Si aparece, entonces se transferirá sin copiar. Si no, entonces el ArrayBuffer será copiado en lugar de transferido. El Ejemplo 15-14 demostrará el uso de esta técnica de transferencia con ArrayBuffers.

### **15.13.6. Mensajería cruzada con postMessage()**

Hay otro caso de uso para el método `postMessage()` en el lado del cliente de JavaScript. Involucra a las ventanas en lugar de los trabajadores, pero hay suficientes similitudes entre los dos casos que vamos a describir el método `postMessage()` del objeto `Window` aquí.

Cuando un documento contiene un elemento `<iframe>`, ese elemento actúa como una ventana incrustada pero independiente. El objeto `Element` que representa el `<iframe>` tiene una propiedad `contentWindow` que es el objeto `Window` para el documento incrustado. Y para los scripts que se ejecutan dentro de ese `iframe` anidado, la propiedad `window.parent` se refiere al objeto `Window` que lo contiene. Cuando dos ventanas muestran documentos con el mismo origen, los scripts de cada una de ellas tienen acceso al contenido de la otra ventana. Pero cuando los documentos tienen orígenes diferentes, la política de mismo origen del navegador impide que el JavaScript de una ventana acceda al contenido de otra.

Para los trabajadores, `postMessage()` proporciona una forma segura para que dos hilos independientes se comuniquen sin compartir memoria. Para las ventanas, `postMessage()` proporciona una forma controlada para que dos orígenes independientes intercambien mensajes de forma segura. Incluso si la política del mismo origen impide que tu script vea el contenido de otra ventana, puedes llamar a `postMessage()` en esa ventana, y al hacerlo se activará un evento "mensaje" en esa ventana, donde puede ser visto por los manejadores de eventos en los scripts de esa ventana.

Sin embargo, el método postMessage() de una Ventana es un poco diferente al método postMessage() de un Trabajador. El primer argumento sigue siendo un mensaje arbitrario que será copiado por el algoritmo de clonación estructurado. Pero el segundo argumento opcional que enumera los objetos que serán transferidos en lugar de copiados se convierte en un tercer argumento opcional. El método postMessage() de una ventana toma una cadena como su segundo argumento requerido. Este segundo argumento debe ser un origen (un protocolo, un nombre de host y un puerto opcional) que especifica quién espera recibir el mensaje. Si pasa la cadena

"<https://good.example.com>" como segundo argumento, pero la ventana a la que está enviando el mensaje contiene realmente contenido de "<https://malware.example.com>", entonces el mensaje que ha enviado no será entregado. Si está dispuesto a enviar su mensaje a contenidos de cualquier origen, entonces puede pasar el comodín "\*" como segundo argumento.

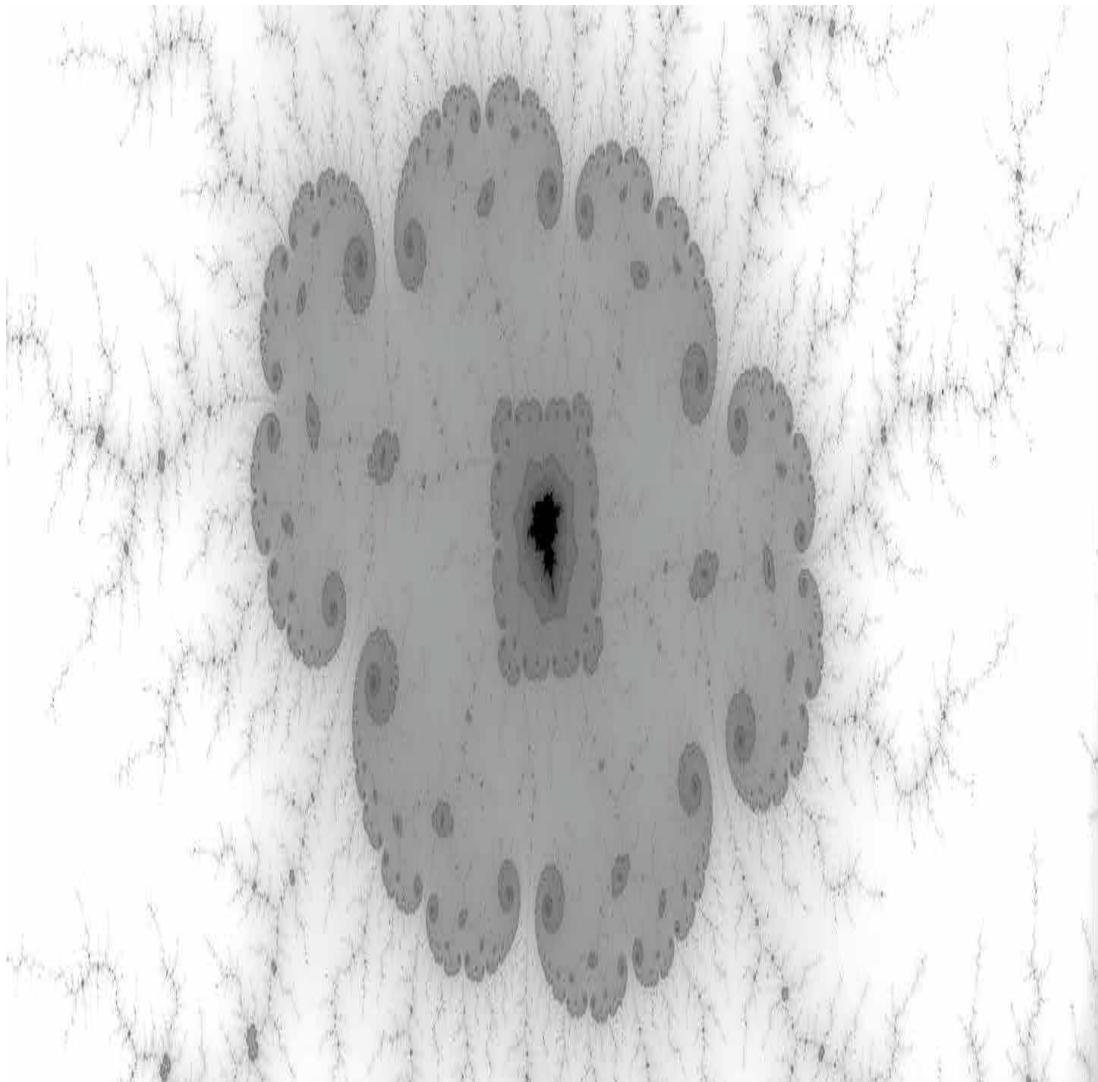
El código JavaScript que se ejecuta dentro de una ventana o <iframe> puede recibir mensajes enviados a esa ventana o marco definiendo la propiedad onmessage de esa ventana o llamando a addEventListener() para los eventos "mensaje". Al igual que con los workers, cuando se recibe un evento "message" para una ventana, la propiedad data del objeto de evento es el mensaje que se envió. Sin embargo, los eventos "mensaje" enviados a las ventanas también definen las propiedades source y origin. La propiedad source especifica el objeto Window que envió el evento, y puede utilizar event.source.postMessage() para enviar una respuesta. La propiedad origin especifica el origen del contenido en la ventana de origen. Esto no es algo que el

remitente del mensaje pueda falsificar, y cuando reciba un evento "mensaje", normalmente querrá verificar que es de un origen que espera.

## 15.14 Ejemplo: El conjunto de Mandelbrot

Este capítulo sobre JavaScript del lado del cliente culmina con un largo ejemplo que demuestra el uso de workers y mensajería para paralelizar tareas computacionalmente intensivas. Pero está escrito para ser una aplicación web atractiva, del mundo real y también demuestra una serie de otras APIs demostradas en este capítulo, incluyendo la gestión del historial; el uso de la clase `ImageData` con un `<canvas>`; y el uso de eventos de teclado, puntero y cambio de tamaño. También demuestra importantes características del núcleo de JavaScript, incluyendo generadores y un uso sofisticado de Promises.

El ejemplo es un programa para visualizar y explorar el conjunto de Mandelbrot, un fractal complejo que incluye bellas imágenes como la que se muestra en [la Figura 15-16](#).



*Figura 15-16. Una parte del conjunto de Mandelbrot*

El conjunto de Mandelbrot se define como el conjunto de puntos del plano complejo que, al ser sometidos a un proceso repetido de multiplicación y suma de complejos, producen un valor cuya magnitud permanece acotada. Los contornos del conjunto son sorprendentemente complejos, y calcular qué puntos son miembros del conjunto y cuáles no lo son es una tarea computacional intensiva: para producir una imagen de  $500 \times 500$  del conjunto de Mandelbrot, hay que calcular individualmente la pertenencia de cada uno de los 250.000 píxeles de la imagen. Y para verificar que el valor asociado a cada píxel sigue estando acotado, puede que tenga que repetir el proceso de

multiplicación compleja 1.000 veces o más. (Un mayor número de iteraciones proporciona unos límites más definidos para el conjunto; un menor número de iteraciones produce unos límites más difusos). Con hasta 250 millones de pasos de aritmética compleja necesarios para producir una imagen de alta calidad del conjunto de Mandelbrot, puede entender por qué el uso de trabajadores es una técnica valiosa. [El ejemplo 15-14](#) muestra el código del trabajador que utilizaremos. Este archivo es relativamente compacto: es sólo el músculo computacional en bruto para el programa más grande. Sin embargo, vale la pena notar dos cosas sobre él:

- El trabajador crea un objeto `ImageData` para representar la cuadrícula rectangular de píxeles para la que está calculando la pertenencia al conjunto de Mandelbrot. Pero en lugar de almacenar los valores reales de los píxeles en el `ImageData`, utiliza una matriz de tipo personalizado para tratar cada píxel como un entero de 32 bits. En esta matriz almacena el número de iteraciones necesarias para cada píxel. Si la magnitud del número complejo calculado para cada píxel es mayor que cuatro, entonces se garantiza matemáticamente que crecerá sin límites a partir de ese momento, y decimos que ha "escapado". Así que el valor que este trabajador devuelve para cada píxel es el número de iteraciones antes de que el valor se haya escapado. Le decimos al trabajador el número máximo de iteraciones que debe intentar para cada valor, y los píxeles que alcanzan este número máximo se consideran

estar en el conjunto.

- El trabajador transfiere el `ArrayBuffer` asociado con el `ImageData` de vuelta al hilo principal para que la memoria asociada a él no tenga que ser copiada.

## Ejemplo 15-14. Código del trabajador para calcular las regiones del conjunto de Mandelbrot

---

```
// Este es un simple trabajador que recibe un mensaje de su hilo padre,  
// realiza el cálculo descrito por ese mensaje y luego publica el // resultado de ese cálculo  
// de vuelta al hilo padre. onmessage = function(message) {  
    // En primer lugar, desempacamos el mensaje recibido:  
    // - el azulejo es un objeto con propiedades de anchura y altura.  
    Especifica la  
    // tamaño del rectángulo de píxeles para el que vamos a calcular  
    // La pertenencia al conjunto de Mandelbrot.  
    // - (x0, y0) es el punto del plano complejo que corresponde al  
    // Píxel superior izquierdo de la baldosa.  
    // - perPixel es el tamaño del píxel en las dimensiones real e imaginaria.  
    // - maxIterations especifica el número máximo de iteraciones que // realizaremos  
    antes de decidir que un píxel está en el conjunto.  
    const {tile, x0, y0, perPixel, maxIterations} = message.data; const  
    {width, height} = tile;  
  
    //A continuación, creamos un objeto ImageData para representar la matriz rectangular  
    // de píxeles, obtener su ArrayBuffer interno, y crear una vista de array tipificada  
    // de ese búfer para que podamos tratar cada píxel como un único número entero en  
    lugar de  
    // cuatro bytes individuales. Almacenaremos el número de iteraciones de cada  
    // píxel en esta matriz de iteraciones. (Las iteraciones se transformarán en // colores de  
    píxeles reales en el hilo padre). const imageData = new ImageData(width, height);
```



```

} i = 2*r*i + y; // Calcula la parte imaginaria de z(n+1).
r = rr - ii + x; // Y la parte real de z(n+1). } iteraciones[index++] = n; // Recuerda
# iteraciones para cada píxel.
if (n > max) max = n; // Registra el número máximo que hemos visto.
if (n < min) min = n; // Y el mínimo también. }
}

// Cuando el cómputo se completa, envía los resultados de vuelta al padre
// hilo. El objeto imageData se copiará, pero el gigantesco ArrayBuffer // que contiene
se transferirá para mejorar el rendimiento.
postMessage({tile, imageData, min, max}, [imageData.data.buffer]);
};

```

La aplicación del visor del conjunto de Mandelbrot que utiliza ese código de trabajador se muestra en [el Ejemplo 15-15](#). Ahora que has llegado casi al final de este capítulo, este largo ejemplo es una especie de experiencia de culminación que reúne una serie de importantes características y APIs de JavaScript del lado del cliente y del núcleo. El código está minuciosamente comentado, y te animo a que lo leas con atención.

### *Ejemplo 15-15. Una aplicación web para visualizar y explorar el conjunto de Mandelbrot*

---

```

/* * Esta clase representa un subrectángulo de un lienzo o imagen.
Utilizamos las baldosas para
* Dividir un lienzo en regiones que puedan ser procesadas independientemente por los
Workers.
*/

```

```

class Tile { constructor(x, y, width, height) {
    this.x = x; // Las propiedades de un objeto Tile this. y = y; // representan la posición y el
    tamaño this. width = width; // de la baldosa dentro de un rectángulo mayor this. height =
    height; //.
}

// Este método estático es un generador que divide un rectángulo de la
// la anchura y la altura especificadas en el número especificado de filas y // columnas y
// produce numRows*numCols Objetos de mosaico para cubrir el rectángulo.
static *tiles(width, height, numRows, numCols) { let columnWidth = Math.
ceil(width / numCols); let rowHeight = Math. ceil(height / numRows);

for(let row = 0; row < numRows; row++) { let tileHeight = (row <
numRows-1)
    ? rowHeight // altura de la mayoría de las filas : height - rowHeight *
(numRows-1); // altura de la última fila for(let col = 0; col < numCols; col++) { let
tileWidth = (col < numCols-1)
    ? columnWidth // anchura de la mayoría de las columnas : width -
columnWidth * (numCols-1); // y última columna

yield new Tile(col * columnWidth, row * rowHeight, tileWidth,
tileHeight); }
}
}
}

```

```

/*
 *      Esta clase representa un conjunto de trabajadores, todos ellos ejecutando el
 * mismo código. La página web
 *      El código del trabajador que especifique debe responder a cada mensaje que
 * reciba mediante
 *          realizar algún tipo de cálculo y luego publicar un único mensaje con
 * el resultado de ese cálculo.
 *
 *      Dado un WorkerPool y un mensaje que representa el trabajo a realizar,
 * simplemente
 *          llamar a addWork(), con el mensaje como argumento. Si hay un Trabajador
 * que se encuentra inactivo, el mensaje se enviará a ese trabajador
 * inmediatamente. Si no hay objetos Worker inactivos, el mensaje se * pondrá en
 * cola y se enviará a un Worker cuando haya uno disponible.
 *
 *      addWork() devuelve una Promise, que se resolverá con el mensaje recibido * del
 * trabajo, o se rechazará si el trabajador lanza un error no manejado. */ class WorkerPool {
constructor(numWorkers, workerSource) { this.idleWorkers = []; // Trabajadores que no
están trabajando actualmente this.workQueue = []; // Trabajos que no están siendo
procesados actualmente this.workerMap = new Map(); // Mapear trabajadores para
resolver y rechazar funciones

// Crea el número especificado de trabajadores, añade manejadores de mensajes y //
errores y los guarda en el array idleWorkers. for(let i = 0; i < numWorkers; i++) { let
worker = new Worker(workerSource); worker.onmessage = message => {
    este._workerDone(worker, null, message.data); }; worker.onerror = error =>
{
    esto._workerDone(worker, error, null);
};
}

```

```
this. idleWorkers[i] = worker; }

// Este método interno es llamado cuando un trabajador termina de trabajar, ya sea
// enviando un mensaje o lanzando un error.
_workerDone(worker, error, response) {
    // Busca las funciones resolve() y reject() para este trabajador // y luego elimina la
    // entrada del trabajador del mapa. let [resolver, rejector] = this. workerMap. get(worker);
    this. workerMap. delete(worker);

    // Si no hay trabajo en cola, vuelve a poner a este trabajador
    // la lista de trabajadores inactivos. En caso contrario, toma trabajo de la cola // y lo
    // envia a este trabajador.
    if (this. workQueue. length === 0) { this. idleWorkers. push(worker); } else { let
        [work, resolver, rejector] = this. workQueue. shift(); this. workerMap. set(worker,
        [resolver, rejector]); worker. postMessage(work); }

    // Finalmente, resuelve o rechaza la promesa asociada al trabajador.
    error === null ? resolver(response) : rejector(error); }

// Este método añade trabajo al pool de trabajadores y devuelve un
Promete que
    // se resolverá con la respuesta de un trabajador cuando el trabajo esté terminado. El
    // trabajo
    // es un valor que se pasa a un trabajador con postMessage().
Si hay un trabajador // inactivo, el mensaje de trabajo se enviará inmediatamente.
De lo contrario
    // se pondrá en cola hasta que haya un trabajador disponible.
addWork(work) { return new Promise((resolve, reject) => {
```

```

        if (this.idleWorkers.length > 0) { let worker = this.idleWorkers.pop(); this.
        workerMap.set(worker, [resolve, reject]); worker.postMessage(work); } else { this.
        workQueue.push([work, resolve, reject]); }

    });

}

/*
*   Esta clase contiene la información de estado necesaria para representar un
conjuntoMandelbrot.
*   Las propiedades cx y cy dan el punto en el plano complejo que es el
*   centro de la imagen. La propiedad perPixel especifica cuánto cambian las partes
real e * imaginaria de ese número complejo para cada píxel de la imagen. La propiedad
maxIterations especifica cuánto trabajamos para calcular el conjunto. Los números más
grandes requieren más cálculos pero producen imágenes más nítidas.
*   Tenga en cuenta que el tamaño del lienzo no forma parte del estado.

Datos cx, cy y
*   porPixel simplemente renderizamos la porción del Mandelbrotset que cabe en
*   el lienzo en su tamaño actual.

*
*   Los objetos de este tipo se utilizan con history.pushState() y sirven para leer
*   el estado deseado desde una URL marcada o compartida.

*/ class PageState {
    //Este método de fábrica devuelve un estado inicial para mostrar todo el conjunto.
    static initialState() { let s = new PageState();
        s.cx = -0.5;
        s.cy = 0;
        s.perPixel = 3/window.innerHeight;
        s.maxIterations = 500; return s;
    }
}

```

```

}

// Este método de fábrica obtiene el estado de una URL, o devuelve null si // no se ha
podido leer un estado válido de la URL. static fromURL(url) { let s = new PageState(); let u =
new URL(url); // Inicializa el estado a partir de los parámetros de búsqueda de la url.

    s.cx = parseFloat(u.searchParams.get("cx"));
    s.cy = parseFloat(u.searchParams.get("cy"));
    s.perPixel = parseFloat(u.searchParams.get("pp"));
    s.maxIterations = parseInt(u.searchParams.get("it"));
    // Si tenemos valores válidos, devuelve el objeto PageState, en caso contrario, null.
    return (isNaN(s.cx) || isNaN(s.cy) | isNaN(s.perPixel)
        || isNaN(s.maxIterations))
        ? null
        : s;
}

// Este método de instancia codifica el estado actual en los parámetros // de búsqueda
de la ubicación actual del navegador. toURL() { let u = new URL(window.location);
    u.searchParams.set("cx", this.cx);
    u.searchParams.set("cy", this.cy);
    u.searchParams.set("pp", this.perPixel);
    u.searchParams.set("it", this.maxIterations); return u.href; }

// Estas constantes controlan el paralelismo del cálculo del conjunto de Mandelbrot. //
Es posible que tengas que ajustarlas para obtener un rendimiento óptimo en tu
ordenador.
const ROWS = 3, COLS = 4, NUMWORKERS = navigator.hardwareConcurrency || 2;

// Esta es la clase principal de nuestro programa de conjuntos de Mandelbrot. Simplemente

```

*invocar el*

```
// función constructora con el elemento <canvas> a renderizar. El programa
// asume que este elemento <canvas> está estilizado para que siempre sea tan //
// grande como la ventana del navegador. class MandelbrotCanvas { constructor(canvas) {
    // Almacenar el lienzo, obtener su objeto contexto, e inicializar un
    WorkerPool this.canvas = canvas; this.context = canvas.getContext("2d"); this.
    workerPool = new WorkerPool(NUMWORKERS, "mandelbrotWorker.js");

    // Define algunas propiedades que usaremos más adelante this.tiles = null; //
    Subregiones del lienzo this.pendingRender = null; // No estamos renderizando
    actualmente this.wantsRerender = false; // No se solicita actualmente ningún renderizado
    this.resizeTimer = null; // Evita que redimensionemos con demasiada frecuencia this.
    colorTable = null; // Para convertir los datos brutos en valores de píxeles.

    // Configurar nuestros manejadores de eventos this.canvas.
    addEventListener("pointerdown", e => this.handlePointer(e)); window.
    addEventListener("keydown", e => this.handleKey(e)); window.
    addEventListener("resize", e => this.handleResize(e)); window.
    addEventListener("popstate", e => this.setState(e.state, false));

    // Inicializar nuestro estado desde la URL o comenzar con el estado inicial.
    this.state = PageState.fromURL(window.location) |||
    PageState.initialState(); // Guardar este estado con el mecanismo del
    historial.
```

```
history.replaceState(this.state, "", this.state.toURL());  
  
// Establecer el tamaño del lienzo y obtener un array de baldosas que lo cubran.  
this.setSize();  
  
// Y renderizar el conjunto de Mandelbrot en el lienzo.  
this.render(); }  
  
// Establecer el tamaño del lienzo e inicializar un array de objetos Tile. Este  
// el método es llamado desde el constructor y también por el método handleResize() //  
// cuando la ventana del navegador es redimensionada. setSize() { this.width = this.canvas.  
width = window.innerWidth; this.height = this.canvas.height = window.innerHeight;  
this.tiles = [... Tile.tiles(this.width, this.height, ROWS, COLS)];  
}  
  
// Esta función hace un cambio en el PageState, y luego vuelve a presentar el  
// Conjunto de Mandelbrot utilizando ese nuevo estado, y también guarda el nuevo estado  
con  
// history.pushState(). Si el primer argumento es una función que la función  
// se llamará con el objeto de estado como argumento y deberá hacer  
// cambios en el estado. Si el primer argumento es un objeto, entonces simplemente  
// copiar las propiedades de ese objeto en el objeto de estado. Si la opción  
// el segundo argumento es falso, entonces el nuevo estado no se guardará. (Nosotros  
// hacer esto cuando se llama a setState en respuesta a un evento popstate.) setState(f,  
save=true) {  
    // Si el argumento es una función, llámala para actualizar el estado.      // En caso  
    contrario, copia sus propiedades en el estado actual.  
    if (typeof f === "function") {
```

```
f(this.state); } else { for(let property in f) { this.state[property] =  
f[property]; }  
  
// En cualquier caso, empieza a renderizar el nuevo estado lo antes posible. this.  
render();  
  
// Normalmente guardamos el nuevo estado. Excepto cuando se nos llama con // un  
segundo argumento de false, lo que hacemos cuando recibimos un evento popstate.  
if (save) { history.pushState(this.state, "", this.state.  
toURL()); }  
  
// Este método dibuja de forma asíncrona la parte del  
Conjunto de Mandelbrot  
// especificado por el objeto PageState en el lienzo. Es llamado por  
// el constructor, por setState() cuando el estado cambia, y por el manejador del evento  
// resize cuando el tamaño del lienzo cambia.  
render() {  
// A veces el usuario puede utilizar el teclado o el ratón para solicitar renders  
// más rápido de lo que podemos realizarlas. No queremos enviar todos los  
// los renders al pool de trabajadores. En cambio, si estamos renderizando, haremos  
// simplemente anota que se necesita un nuevo render, y cuando el actual  
// se completa el renderizado, renderizaremos el estado actual, posiblemente  
// saltando // múltiples estados intermedios. if (this.pendingRender) { // Si ya estamos  
renderizando, this.wantsRerender = true; // toma nota para rerenderizar más tarde
```

```
    return; // y no hagas nada más ahora.      }

    // Obtener nuestras variables de estado y calcular el número complejo para la //
    esquina superior izquierda del lienzo. let {cx, cy, perPixel, maxIterations} = this.state;
    let x0 = cx - perPixel * this.width/2; let y0 = cy - perPixel * this.height/2;

    // Para cada uno de nuestros mosaicos ROWS*COLS, llame a addWork() con un mensaje //
    // para el código en mandelbrotWorker.js. Recoger los objetos Promise //
    resultantes en un array. let promises = this.tiles.map(tile => this.workerPool.
    addWork({ tile: tile, x0: x0 + tile.x * perPixel, y0: y0 + tile.y * perPixel, perPixel:
    perPixel, maxIterations: maxIterations }));

    // Usa Promise.all() para obtener un array de respuestas del array de //
    promesas. Cada respuesta es el cálculo de una de nuestras fichas.
    // Recordemos de mandelbrotWorker.js que cada respuesta incluye el
    // El objeto Tile, un objeto ImageData que incluye los recuentos de iteración
    // en lugar de los valores de los píxeles, y las iteraciones mínimas y máximas // para
    ese azulejo.

    this.pendingRender = Promise.all(promises).then(responses => {

        // Primero, encuentra las iteraciones máximas y mínimas globales sobre todos los
        azulejos.      // Necesitamos estos números para poder asignar colores a los píxeles.
        dejar min = maxIterations, max = 0;
```

```

for(let r of responses) { if (r. min < min) min = r. min; if (r.
max > max) max = r. max; }

// Ahora necesitamos una manera de convertir los recuentos de iteración crudos de
los // trabajadores en colores de píxeles que se mostrarán en el lienzo.
// Sabemos que todos los píxeles tienen entre mínimo y máximo de iteraciones
// así que precalculamos los colores para cada cuenta de iteración y los
almacenamos // en el array colorTable.

// Si aún no hemos asignado una tabla de colores, o si ya no tiene // el tamaño
adecuado, entonces asigna una nueva.
if (! this. colorTable || this. colorTable. length !== maxIterations+1){
    this. colorTable = new Uint32Array(maxIterations+1);
}

// Dados el máximo y el mínimo, calcular los valores apropiados en el
// tabla de colores. Los píxeles del conjunto se colorearán de forma totalmente opaca
// negro. Los píxeles fuera del conjunto serán negros translúcidos con mayor
// el recuento de iteraciones da lugar a una mayor opacidad.

Píxeles con
// los recuentos mínimos de iteración serán transparentes y el fondo // blanco se
mostrará a través, dando como resultado una imagen en escala de grises.
if (min === max) { // Si todos los píxeles son iguales, if (min === maxIterations) { //
Entonces hazlos todos negros this. colorTable[min] = 0xFF000000;
} else { // O todo transparente.
    this. colorTable[min] = 0;
}

```

```

        }
    } si no {
        // En el caso normal de que el mínimo y el máximo sean diferentes, utilice un
        // escala logarítmica para asignar a cada posible recuento de iteraciones un
        // opacidad entre 0 y 255, y luego utilizar el operador shift left // para
        // convertirlo en un valor de pixel.
        let maxlog = Math. log(1+max-min); for(let i = min; i <= max; i++) { this.
colorTable[i] = (Math. ceil(Math. log(1+i-min)/maxlog *
255) << 24);
        }
    }

    // Ahora traduce los números de iteración en los // imageData de cada
    respuesta a colores de la colorTable.
    for(let r of responses) { let iterations = new Uint32Array(r. imageData. data.
buffer); for(let i = 0; i < iterations. length; i++) { iterations[i] = this.
colorTable[iterations[i]]; }
}

    // Finalmente, renderiza todos los objetos imageData en sus // correspondientes
    mosaicos del lienzo usando putImageData().
    (Primero, sin embargo, elimine cualquier transformación CSS en el lienzo que pueda
    // haber sido establecida por el controlador del evento pointerdown). this. canvas. style.
    transform = ""; for(let r of responses) { this. context. putImageData(r. imageData,
r. tile. x, r. tile. y); }
})

```

```
. catch((reason) => {
  // Si algo va mal en alguna de nuestras promesas, registraremos
  // un error aquí. Esto no debería suceder, pero esto ayudará con // la depuración
  si lo hace.
  console.error("Promesa rechazada en render()", reason);
  . finally(() => {
    // Cuando hayamos terminado de renderizar, borra las banderas de
    pendingRender this. pendingRender = null;
    // Y si las peticiones de renderización llegaron mientras estábamos ocupados,
    rerenderiza ahora.
    if (this. wantsRerender) { this. wantsRerender = false;
      this. render();
    });
  }
}

// Si el usuario cambia el tamaño de la ventana, esta función será llamada
repetidamente.
// Redimensionar un lienzo y volver a renderizar el conjunto de Mandelbrot es una tarea
costosa
// operación que no podemos hacer varias veces por segundo, por lo que utilizamos un
temporizador
// para aplazar el manejo del redimensionamiento hasta que hayan transcurrido
200ms desde que se recibió el último // evento de redimensionamiento.
handleResize(event) { // Si ya estábamos aplazando un redimensionamiento, borralo.
if (this. resizeTimer) clearTimeout(this. resizeTimer); // Y aplaza este
redimensionamiento en su lugar.
  this. resizeTimer = setTimeout(() => { this. resizeTimer = null; // Observe que el
redimensionamiento ha sido manejado this. setSize(); // Redimensiona el lienzo y los
azulejos this. render(); // Rerenderiza con el nuevo tamaño
}, 200);
```

```
}
```

```
// Si el usuario pulsa una tecla, se llamará a este manejador de eventos.  
// Llamamos a setState() en respuesta a varias claves, y setState() renderiza // el  
nuevo estado, actualiza la URL, y guarda el estado en el historial del navegador.  
handleKey(event) { switch(event.key) { case "Escape": // Escribe Escape para volver  
al estado inicial this. setState(PageState.initialState()); break; case "+": // Escribe +  
para aumentar el número de iteraciones this. setState(s => {  
    s.maxIterations = Math.round(s.maxIterations*1.5); }); break; case "-": //  
Escribe - para disminuir el número de iteraciones this. setState(s => {  
    s.maxIterations = Math.round(s.maxIterations/1.5);  
    if (s.maxIterations < 1) s.maxIterations = 1; }); break; case "o": // Escribe  
o para alejar el zoom this. setState(s => s.perPixel *= 2); break; case "ArrowUp": //  
Flecha arriba para desplazarse hacia arriba this. setState(s => s.cy -= this.height/10 * s.  
perPixel); break; case "ArrowDown": // Flecha hacia abajo para desplazarse hacia  
abajo this. setState(s => s.cy += this.height/10 * s.perPixel); break; case "ArrowLeft":  
// Flecha izquierda para desplazarse hacia la izquierda this. setState(s => s.cx -= this.  
width/10 *
```

```
s. perPixel); break; case "ArrowRight": // Flecha derecha para desplazarse a la
derecha this. setState(s => s. cx += this. width/10 * s. perPixel); break; }
}

//Este método es llamado cuando recibimos un evento de puntero hacia abajo en el lienzo.
//El evento pointerdown podría ser el inicio de un gesto de zoom (un clic o
//toque) o un gesto de desplazamiento (un arrastre). Este manejador registra
manejadores para
//los eventos pointermove y pointerup para responder al resto
//del gesto. (Estos dos manejadores extra se eliminan cuando el gesto // termina
con un pointerup.) handlePointer(event) {
    // Las coordenadas de los píxeles y el tiempo del puntero inicial hacia abajo.
    // Como el lienzo es tan grande como la ventana, estas coordenadas de evento //
son también coordenadas del lienzo. const x0 = event. clientX, y0 = event. clientY, t0 =
Date. now();

    // Este es el manejador de los eventos de movimiento. const
pointerMoveHandler = event => {
    //¿Cuánto hemos avanzado y cuánto tiempo ha pasado?
    let dx=evento. clienteX-x0, dy=evento. clienteY-y0, dt=Date. now()-t0;

    //Si el puntero se ha movido lo suficiente o ha pasado el tiempo suficiente como para
que // no se trate de un clic normal, entonces usa CSS para desplazar la pantalla.
    // (Lo renderizaremos de verdad cuando recibamos el evento pointerup.) if (dx >
10 || dy > 10 || dt > 500) { this. canvas. style. transform =
```

```
`translate(${dx}px, ${dy}px)`;  
}  
};  
  
// Este es el manejador de los eventos pointerup const pointerUpHandler =  
evento => {  
    // Cuando el puntero sube, el gesto ha terminado, así que elimina // los  
    // manejadores de movimiento y subida hasta el siguiente gesto.  
    this.canvas.removeEventListener("pointermove", pointerMoveHandler);  
    this.canvas.removeEventListener("pointerup", pointerUpHandler);  
  
    // ¿Cuánto se ha movido el puntero y cuánto tiempo ha pasado?  
    const dx = evento.clientX-x0, dy=evento.clientY-y0, dt=Date.now()-t0;  
    // Descomponer el objeto de estado en constantes individuales. const  
{cx, cy, perPixel} = this.state;  
  
    // Si el puntero se ha movido lo suficiente o si ha pasado el tiempo suficiente, entonces  
    // esto fue un gesto de pan, y necesitamos cambiar de estado para cambiar  
    // el punto central. De lo contrario, el usuario hizo clic o tocó en un // punto y  
    // necesitamos centrar y hacer zoom en ese punto.  
    si (dx > 10 || dy > 10 || dt > 500) {  
        // El usuario ha desplazado la imagen en (dx, dy) píxeles.           // Convierte  
        // esos valores en desplazamientos en el plano complejo. this.setState({cx: cx - dx*perPixel,  
        cy: cy - dy*perPixel});  
    } si no {  
        // El usuario hizo clic. Calcula cuántos píxeles se mueve el centro.  
        let cdx = x0 - this.width/2; let cdy = y0 - this.height/2;  
  
        // Utiliza CSS para ampliar la imagen de forma rápida y temporal
```

```
this.canvas.style.transform = `translate(${-cdx*2}px, ${-cdy*2}px)  
scale(2)`;  
  
// Establecer las coordenadas complejas del nuevo punto central y //  
ampliarlo en un factor de 2. this.setState(s => {  
    s.cx += cdx * s.perPixel;  
    s.cy += cdy * s.perPixel;  
    s.perPixel /= 2;});  
}  
};  
  
// Cuando el usuario comienza un gesto, registramos los manejadores para los  
eventos // pointermove y pointerup que siguen.  
this.canvas.addEventListener("pointermove", pointerMoveHandler);  
this.canvas.addEventListener("pointerup", pointerUpHandler); }  
}  
  
// Por último, así es como configuramos el lienzo. Tenga en cuenta que este  
Archivo JavaScript  
// es autosuficiente. El archivo HTML sólo necesita incluir este <script>. let canvas =  
document.createElement("canvas"); // Crear un elemento canvas document.body.  
append(canvas); // Insertarlo en el documento body. body.style = "margin:0"; // Sin  
margen para el <body> canvas.style.width = "100%"; // Hacer el canvas tan ancho como  
el body. style.height = "100%"; // y tan alto como el body.  
new MandelbrotCanvas(canvas); // ¡Y empieza a renderizar en él!
```

## 15.15 Resumen y sugerencias de lecturas adicionales

Este largo capítulo ha cubierto los fundamentos de la programación JavaScript del lado del cliente:

- Cómo se incluyen los scripts y los módulos de JavaScript en las páginas web y cómo y cuándo se ejecutan.
- El modelo de programación asíncrona y basada en eventos de JavaScript del lado del cliente.
- El Modelo de Objetos del Documento (DOM) que permite al código JavaScript inspeccionar y modificar el contenido HTML del documento en el que está incrustado. Esta API DOM es el corazón de toda la programación JavaScript del lado del cliente.
- Cómo el código JavaScript puede manipular los estilos CSS que se aplican al contenido dentro del documento.
- Cómo el código JavaScript puede obtener las coordenadas de los elementos del documento en la ventana del navegador y dentro del propio documento.
- Cómo crear "componentes web" de interfaz de usuario reutilizables con JavaScript, HTML y CSS utilizando las APIs de Custom Elements y Shadow DOM.
- Cómo mostrar y generar dinámicamente gráficos con SVG y el elemento HTML <canvas>.
- Cómo añadir efectos de sonido con guión (tanto grabados como sintetizados) a tus páginas web.
- Cómo JavaScript puede hacer que el navegador cargue nuevas páginas, retroceda y avance en el historial de navegación del usuario, e incluso añada nuevas entradas al historial de navegación.

- Cómo los programas JavaScript pueden intercambiar datos con los servidores web utilizando los protocolos HTTP y WebSocket.
- Cómo los programas de JavaScript pueden almacenar datos en el navegador del usuario.

Cómo los programas de JavaScript pueden utilizar hilos de trabajo para lograr una forma segura de concurrencia.

Este ha sido el capítulo más largo del libro, con diferencia. Pero no puede acercarse a cubrir todas las APIs disponibles para los navegadores web. La plataforma web es extensa y está en constante evolución, y mi objetivo para este capítulo era presentar las APIs más importantes. Con el conocimiento que tienes de este libro, estás bien equipado para aprender y usar nuevas APIs cuando las necesites. Pero no puedes aprender sobre una nueva API si no sabes que existe, así que las breves secciones que siguen terminan el capítulo con una lista rápida de características de la plataforma web que podrías querer investigar en el futuro.

### **15.15.1 HTML y CSS**

La web se basa en tres tecnologías clave: HTML, CSS y JavaScript, y el conocimiento de JavaScript sólo puede llevarte hasta cierto punto como desarrollador web a menos que también desarrolles tu experiencia con HTML y CSS. Es importante saber cómo utilizar JavaScript para manipular elementos HTML y estilos CSS, pero ese conocimiento es mucho más útil si también sabes qué elementos HTML y qué estilos CSS utilizar.

Así que antes de empezar a explorar más APIs de JavaScript, te animo a que inviertas algo de tiempo en dominar las otras

herramientas de un desarrollador web. Los formularios y elementos de entrada de HTML, por ejemplo, tienen un comportamiento sofisticado que es importante entender, y los modos de diseño flexbox y grid en CSS son increíblemente potentes.

Dos temas a los que merece la pena prestar especial atención en este ámbito son la accesibilidad (incluidos los atributos ARIA) y la internacionalización (incluida la compatibilidad con las direcciones de escritura de derecha a izquierda).

### 15.15.2 Rendimiento

Una vez que has escrito una aplicación web y la has lanzado al mundo, comienza la interminable búsqueda para hacerla rápida. Sin embargo, es difícil optimizar cosas que no se pueden medir, por lo que vale la pena familiarizarse con las API de rendimiento. La propiedad performance del objeto ventana es el principal punto de entrada a esta API. Incluye una fuente de tiempo de alta resolución `performance.now()`, y los métodos `performance.mark()` y `performance.measure()` para marcar puntos críticos en su código y midiendo el tiempo transcurrido entre ellos. Al llamar a estos métodos se crean objetos `PerformanceEntry` a los que puedes acceder con `performance.getEntries()`. Los navegadores añaden sus propios objetos `PerformanceEntry` cada vez que el navegador carga una nueva página o recupera un archivo a través de la red, y estos objetos `PerformanceEntry` creados automáticamente incluyen detalles de tiempo granulares del rendimiento de la red de tu aplicación. Los objetos

La clase `PerformanceObserver` permite especificar una función que se invoca cuando se crean nuevos objetos `PerformanceEntry`.

### 15.15.3 Seguridad

Este capítulo ha introducido la idea general de cómo defenderse de las vulnerabilidades de seguridad del cross-site scripting (XSS) en sus sitios web, pero no hemos entrado en muchos detalles. El tema de la seguridad en la web es importante, y es posible que quieras dedicar algo de tiempo a aprender más sobre él. Además de XSS, vale la pena aprender sobre la cabecera HTTP `ContentSecurity-Policy` y entender cómo CSP le permite pedir al navegador web que restrinja las capacidades que concede a código JavaScript. También es importante entender el CORS (Cross-Origin Resource Sharing).

### 15.15.4 WebAssembly

WebAssembly (o "wasm") es un formato de bytecode de máquina virtual de bajo nivel que está diseñado para integrarse bien con los intérpretes de JavaScript en los navegadores web. Existen compiladores que permiten compilar programas C, C++ y Rust a bytecode WebAssembly y ejecutar esos programas en los navegadores web a una velocidad cercana a la nativa, sin romper el sandbox del navegador o el modelo de seguridad. WebAssembly puede exportar funciones que pueden ser llamadas por programas de JavaScript. Un caso típico de uso de WebAssembly sería compilar la biblioteca de compresión zlib estándar del lenguaje C para que el código JavaScript tenga

acceso a algoritmos de compresión y descompresión de alta velocidad. Más información en <https://webassembly.org>.

### 15.15.5 Más funciones de documentos y ventanas

Los objetos Ventana y Documento tienen una serie de características que no se han tratado en este capítulo:

- El objeto Window define los métodos alert(), confirm() y prompt() que muestran diálogos modales simples al usuario. Estos métodos bloquean el hilo principal. El método confirm() devuelve sincrónicamente un valor booleano, y prompt() devuelve sincrónicamente una cadena de entrada del usuario. No son adecuados para su uso en producción, pero pueden ser útiles para proyectos y prototipos sencillos.
- Las propiedades del navegador y de la pantalla de la ventana se mencionaron de pasada al principio de este capítulo, pero los objetos Navigator y Screen a los que hacen referencia tienen algunas características que no se describieron aquí y que pueden resultarle útiles.
- El método requestFullscreen() de cualquier objeto Element solicita que ese elemento (un elemento <video> o <canvas>, por ejemplo) se muestre en modo de pantalla completa. El método exitFullscreen() del documento vuelve al modo de visualización normal.
- El método requestAnimationFrame() de la ventana toma una función como argumento y ejecutará esa función cuando el navegador se prepare para renderizar el siguiente cuadro. Cuando se realizan cambios visuales (especialmente los repetidos o animados), envolver el código dentro de una llamada a requestAnimationFrame() puede ayudar a asegurar que los

cambios se rendericen sin problemas y de forma optimizada por el navegador.

- Si el usuario selecciona un texto dentro de su documento, puede obtener detalles de esa selección con el método `getSelection()` de `Window` y obtener el texto seleccionado con `getSelection().toString()`. En algunos navegadores, `navigator.clipboard` es un objeto con una API asíncrona para leer y fijar el contenido del portapapeles del sistema para permitir interacciones de copiar y pegar con aplicaciones fuera del navegador.
- Una característica poco conocida de los navegadores web es que los elementos HTML con un atributo `contenteditable="true"` permiten editar su contenido. La página `El método document.execCommand() habilita las funciones de edición de texto enriquecido para el contenido editable.`
- Un `MutationObserver` permite a JavaScript monitorizar los cambios que se producen en un elemento específico del documento, o por debajo de él. Cree un `MutationObserver` con la función `MutationObserver()` pasando la función `callback` que debe ser llamada cuando se realicen cambios. A continuación, llame al método `observe()` del `MutationObserver` para especificar qué partes de qué elemento deben ser monitoreadas.
- Un `IntersectionObserver` permite a JavaScript determinar qué elementos del documento están en la pantalla y cuáles están cerca de estarlo. Resulta especialmente útil para las aplicaciones que quieren cargar dinámicamente el contenido bajo demanda a medida que el usuario se desplaza.

## 15.15.6 Eventos

El gran número y la diversidad de eventos que admite la plataforma web pueden resultar desalentadores. En este capítulo se han analizado diversos tipos de eventos, pero aquí hay algunos más que pueden resultarle útiles:

- Los navegadores disparan eventos "online" y "offline" en el objeto Window cuando el navegador obtiene o pierde una conexión a Internet.
- Los navegadores disparan un evento "visibilitychange" en el objeto Document cuando un documento se vuelve visible o invisible (normalmente porque el usuario ha cambiado de pestaña). JavaScript puede comprobar document.visibilityState para determinar si su documento es actualmente "visible" u "oculto".
- Los navegadores admiten una complicada API para soportar la función de arrastrar y soltar

UIs y para apoyar el intercambio de datos con aplicaciones fuera del navegador. Esta API implica una serie de eventos, como "dragstart", "dragover", "dragend" y "drop". Esta API es complicada de utilizar correctamente, pero es útil cuando se necesita. Es una API importante que debes conocer si quieras que los usuarios puedan arrastrar archivos desde su escritorio a tu aplicación web.

- La API de bloqueo del puntero permite a JavaScript ocultar el puntero del ratón y obtener los eventos crudos del ratón como cantidades de movimiento relativas en lugar de posiciones absolutas en la pantalla. Esto suele ser útil para los juegos. Llame a requestPointerLock() en el elemento al que desea que se dirijan todos los eventos del ratón. Después de hacer esto, los eventos "mousemove" entregados a ese elemento tendrán las propiedades movementX y movementY.

- La API Gamepad añade soporte para los mandos de juego. Utilice `navigator.getGamepads()` para obtener objetos Gamepad conectados, y escuche los eventos "gamepadconnected" en el objeto Window para ser notificado cuando se conecte un nuevo controlador. El objeto Gamepad define una API para consultar el estado actual de los botones del mando.

## 15.15.7 Aplicaciones web progresivas y Service Workers

El término *Aplicaciones Web Progresivas*, o PWAs, es una palabra de moda que describe las aplicaciones web que se construyen utilizando algunas tecnologías clave. La documentación cuidadosa de estas tecnologías clave requeriría un libro propio, y no las he cubierto en este capítulo, pero deberías conocer todas estas APIs. Vale la pena señalar que las potentes y modernas APIs como éstas están normalmente diseñadas para trabajar sólo en conexiones seguras HTTPS. Los sitios web que todavía utilizan URLs `http://` no podrán aprovecharlas:

- Un ServiceWorker es un tipo de hilo de trabajo con la capacidad de interceptar, inspeccionar y responder a las peticiones de red de la aplicación web a la que "da servicio". Cuando una aplicación web registra un service worker, el código de ese trabajador se vuelve persistente en el almacenamiento local del navegador, y cuando el usuario vuelve a visitar el sitio web asociado, el service worker se reactiva. Los service workers pueden almacenar en caché las respuestas de la red (incluidos los archivos de código JavaScript), lo que significa que las aplicaciones web que utilizan service workers pueden instalarse de forma efectiva en el ordenador del usuario para su rápida puesta en marcha y uso sin conexión. El *Service Worker Cookbook* en <https://serviceworke.rs> es un valioso recurso para aprender sobre los service workers y sus tecnologías relacionadas.

- La API de caché está diseñada para ser utilizada por los trabajadores de servicios (pero también está disponible para el código JavaScript normal fuera de los trabajadores). Funciona con los objetos Request y Response definidos por la API fetch() e implementa una caché de pares Request/Response. La API de caché permite a un service worker almacenar en caché los scripts y otros activos de la aplicación web que sirve y también puede ayudar a permitir el uso sin conexión de la aplicación web (lo que es particularmente importante para los dispositivos móviles).
- Un Web Manifest es un archivo con formato JSON que describe una aplicación web, incluyendo un nombre, una URL y enlaces a iconos de varios tamaños. Si tu aplicación web utiliza un service worker e incluye una etiqueta <link rel="manifest"> que hace referencia a un archivo .webmanifest, los navegadores (en particular los navegadores de los dispositivos móviles) pueden darte la opción de añadir un ícono de la aplicación web a tu escritorio o pantalla de inicio.
- La API de notificaciones permite a las aplicaciones web mostrar notificaciones utilizando el sistema de notificaciones nativo del sistema operativo, tanto en dispositivos móviles como de escritorio. Las notificaciones pueden incluir una imagen y un texto, y tu código puede recibir un evento si el usuario hace clic en la notificación. El uso de esta API se complica por el hecho de que primero hay que solicitar el permiso del usuario para mostrar las notificaciones.
- La API Push permite a las aplicaciones web que cuentan con un service worker (y que tienen el permiso del usuario) suscribirse a las notificaciones de un servidor, y mostrar esas notificaciones incluso cuando la propia aplicación no se está ejecutando. Las notificaciones push son habituales en los dispositivos móviles, y la API Push acerca las aplicaciones web a la paridad de características con las aplicaciones nativas en los móviles.

## 15.15.8 API para dispositivos móviles

Hay una serie de APIs web que son principalmente útiles para las aplicaciones web que se ejecutan en dispositivos móviles. (Por desgracia, varias de estas API solo funcionan en dispositivos Android y no en dispositivos iOS).

- La API de geolocalización permite a JavaScript (con el permiso del usuario) determinar la ubicación física del usuario. Es muy compatible con los dispositivos de escritorio y móviles, incluidos los dispositivos iOS. Utilice `navigator.geolocation.getCurrentPosition()` para solicitar la posición actual del usuario y utilizar `navigator.geolocation.watchPosition()` a registrar una llamada de retorno para ser llamada cuando la posición del usuario cambie.
- El método `navigator.vibrate()` hace que un móvil (pero no iOS) para que vibre. A menudo esto solo se permite en respuesta a un gesto del usuario, pero llamar a este método permitirá a tu aplicación proporcionar una respuesta silenciosa de que se ha reconocido un gesto.
- La API ScreenOrientation permite a una aplicación web consultar la orientación actual de la pantalla de un dispositivo móvil y también para bloquearse en la orientación horizontal o vertical.
- Los eventos "devicemotion" y "deviceorientation" del objeto ventana informan de los datos del acelerómetro y el magnetómetro del dispositivo, lo que permite determinar cómo se acelera el dispositivo y cómo lo orienta el usuario en el espacio. (Estos eventos sí funcionan en iOS).
- La API de sensores aún no es ampliamente soportada más allá de Chrome en los dispositivos Android, pero permite el acceso de JavaScript a todo el conjunto de sensores de los dispositivos móviles, incluidos el acelerómetro, el giroscopio, el magnetómetro y el sensor de luz ambiental. Estos sensores

permiten a JavaScript determinar la dirección en la que mira el usuario o detectar cuando el usuario agita su teléfono, por ejemplo.

### 15.15.9 APIs binarias

Las matrices tipificadas, los ArrayBuffers y la clase DataView (todo ello tratado en §11.2) permiten a JavaScript trabajar con datos binarios. Como se ha descrito anteriormente en este capítulo, la API fetch() permite a los programas de JavaScript cargar datos binarios a través de la red. Otra fuente de datos binarios son los archivos del sistema de archivos local del usuario. Por razones de seguridad, JavaScript no puede leer únicamente archivos locales. Pero si el usuario selecciona un archivo para cargarlo (usando un `<input type="file">` form element) o utiliza la función de arrastrar y soltar para soltar un archivo en su aplicación web, entonces JavaScript puede acceder a ese archivo como un objeto File.

File es una subclase de Blob, y como tal, es una representación opaca de un trozo de datos. Puedes utilizar una clase FileReader para obtener de forma asíncrona el contenido de un archivo como un ArrayBuffer o una cadena. (En algunos navegadores, puedes omitir el FileReader y en su lugar utilizar los métodos `text()` y `arrayBuffer()` basados en promesas y definidos por la clase Blob, o el método `stream()` para el acceso en streaming al contenido del archivo).

Cuando se trabaja con datos binarios, especialmente con la transmisión de datos binarios, puede ser necesario decodificar bytes en texto o codificar texto como bytes. Las clases TextEncoder y TextDecoder ayudan en esta tarea.

### **15.15.10 APIs de medios de comunicación**

La función `navigator.mediaDevices.getUserMedia()` permite a JavaScript solicitar el acceso al micrófono y/o a la cámara de vídeo del usuario. Una solicitud exitosa resulta en un objeto `MediaStream`. Los flujos de vídeo pueden mostrarse en una etiqueta `<video>` (estableciendo la propiedad `srcObject` al flujo). Los fotogramas del vídeo pueden ser capturados en un `<canvas>` fuera de la pantalla con la función `canvas.drawImage()` dando como resultado una fotografía de relativamente baja resolución. Los flujos de audio y vídeo devueltos por `getUserMedia()` pueden ser grabados y codificados en un `Blob` con un objeto `MediaRecorder`.

La API WebRTC, más compleja, permite la transmisión y recepción de `MediaStreams` a través de la red, posibilitando, por ejemplo, las videoconferencias peer-to-peer.

### **15.15.11 Criptografía y APIs relacionadas**

La propiedad `crypto` del objeto `Window` expone un método `getRandomValues()` para obtener números pseudoaleatorios criptográficamente seguros. Otros métodos para el cifrado, descifrado, generación de claves, firmas digitales, etc. están disponibles a través de `crypto.subtle`. El nombre de esta propiedad es una advertencia para todos los que utilizan estos métodos de que el uso adecuado de los algoritmos criptográficos es difícil y que no debe utilizar esos métodos a menos que realmente sepa lo que está haciendo. Además, los métodos de `crypto.subtle` sólo están disponibles para el código JavaScript que se ejecuta dentro

de los documentos que fueron cargados a través de una conexión segura HTTPS.

La API de gestión de credenciales y la API de autenticación web permiten que JavaScript genere, almacene y recupere credenciales de clave pública (y de otros tipos) y permite la creación de cuentas y el inicio de sesión sin contraseñas. La API de JavaScript consiste principalmente en las funciones `navigator.credentials.create()` y `navigator.credentials.get()`, pero se requiere una importante infraestructura en el lado del servidor para que estos métodos funcionen. Estas APIs aún no están soportadas universalmente, pero tienen el potencial de revolucionar la forma en que nos conectamos a los sitios web.

La API de solicitud de pago añade soporte al navegador para realizar pagos con tarjeta de crédito en la web. Permite a los usuarios almacenar sus datos de pago de forma segura en el navegador para que no tengan que escribir el número de su tarjeta de crédito cada vez que realicen una compra. Las aplicaciones web que desean solicitar un pago crean un objeto `PaymentRequest` y llaman a su método `show()` para mostrar la solicitud al usuario.

---

Las ediciones anteriores de este libro contaban con una amplia sección de referencia que cubría el JavaScript

<sup>1</sup> biblioteca estándar y APIs web. Se eliminó en la séptima edición porque MDN lo ha hecho obsoleto: hoy en día, es más rápido buscar algo en MDN que hojear un libro, y mis antiguos colegas de MDN hacen un mejor trabajo para mantener su documentación en línea actualizada de lo que este libro jamás podría.

Algunas fuentes, incluida la especificación HTML, hacen una distinción técnica entre

<sup>2</sup> manejadores y oyentes, en función de la forma en que se registran. En este libro, tratamos los dos términos como sinónimos.

Si has utilizado el framework React para crear interfaces de usuario del lado del cliente, esto puede

<sup>3</sup>te sorprenda. React realiza una serie de pequeños cambios en el modelo de eventos del lado del cliente, y uno de ellos es que en React, los nombres de las propiedades de los manejadores de eventos se escriben en camelCase: onClick, onMouseOver, etc. Sin embargo, cuando se trabaja con la plataforma web de forma nativa, las propiedades de los manejadores de eventos se escriben completamente en minúsculas.

La especificación de elementos personalizados permite subclasicar <button> y otros específicos

<sup>4</sup>pero esto no es compatible con Safari y se requiere una sintaxis diferente para utilizar un elemento personalizado que extienda cualquier cosa que no sea HTMLElement.



# Capítulo 16. JavaScript del lado del servidor con Node

---

Node es JavaScript con enlaces al sistema operativo subyacente, lo que hace posible escribir programas de JavaScript que lean y escriban archivos, ejecuten procesos hijos y se comuniquen a través de la red. Esto hace que Node sea útil como:

- Alternativa moderna a los scripts de shell que no sufre la sintaxis arcana de bash y otros shells de Unix.
- Lenguaje de programación de propósito general para ejecutar programas de confianza, no sujeto a las restricciones de seguridad impuestas por los navegadores web al código no fiable.
- Entorno popular para escribir servidores web eficientes y altamente concurrentes.

La característica que define a Node es su concurrencia basada en eventos de un solo hilo, habilitada por una API asíncrona por defecto. Si has programado en otros lenguajes pero no has hecho mucha codificación en JavaScript, o si eres un programador de JavaScript del lado del cliente con experiencia y acostumbrado a escribir código para navegadores web, usar Node será un poco de ajuste, como lo es cualquier lenguaje o entorno de programación nuevo. Este capítulo comienza explicando el modelo de programación de Node, con énfasis en la concurrencia, la API de Node para trabajar con datos en streaming, y el tipo Buffer de Node para trabajar con datos

binarios. Estas secciones iniciales son seguidas por secciones que destacan y demuestran algunas de las APIs más importantes de Node, incluyendo aquellas para trabajar con archivos, redes, procesos e hilos.

Un capítulo no es suficiente para documentar todas las APIs de Node, pero mi esperanza es que este capítulo explique lo suficiente de los fundamentos para que seas productivo con Node, y confíes en que puedes dominar cualquier nueva API que

### INSTALACIÓN DEL NODO

Node es un software de código abierto. Visite <https://nodejs.org> para descargar e instalar Node para Windows y MacOS. En Linux, puedes instalar Node con tu gestor de paquetes normal, o puedes visitar <https://nodejs.org/en/download> para descargar los binarios directamente. Si trabajas con software en contenedores, puedes encontrar imágenes oficiales de Node para Docker en <https://hub.docker.com>.

Además del ejecutable de Node, una instalación de Node también incluye npm, un gestor de paquetes que permite un fácil acceso a un vasto ecosistema de herramientas y librerías JavaScript. Los ejemplos de este capítulo utilizarán únicamente los paquetes incorporados de Node y no requerirán npm ni ninguna librería externa.

Por último, no pase por alto la documentación oficial de Node, disponible en <https://nodejs.org/api> y <https://nodejs.org/docs/guides>. Lo he encontrado bien organizado y bien escrito.

necesites.

## 16.1 Fundamentos de la programación de nodos

Comenzaremos este capítulo con un rápido vistazo a cómo se estructuran los programas Node y cómo interactúan con el sistema operativo.

### 16.1.1 Salida de la consola

Si estás acostumbrado a programar en JavaScript para navegadores web, una de las pequeñas sorpresas de Node es que `console.log()` no sólo sirve para depurar, sino que es la forma más sencilla de Node de mostrar un mensaje al usuario o, más generalmente, de enviar la salida al flujo `stdout`. Aquí está el clásico programa "Hola Mundo" en Node:

```
console.log("¡Hola Mundo!");
```

Hay formas de nivel inferior para escribir en `stdout`, pero no hay una forma más elegante u oficial que simplemente llamar a `console.log()`.

En los navegadores web, `console.log()`, `console.warn()` y `console.error()` suelen mostrar pequeños iconos junto a su salida en la consola del desarrollador para indicar la variedad del mensaje de registro. Node no hace esto, pero la salida mostrada con `console.error()` se distingue de la salida mostrada con `console.log()` porque `console.error()` escribe en el flujo `stderr`. Si está utilizando Node para escribir un programa que está diseñado para tener `stdout` redirigido a un archivo o a una tubería, puede utilizar `console.error()` para mostrar el texto a la consola donde el

usuario lo verá, aunque el texto impreso con `console.log()` esté oculto.

### 16.1.2 Argumentos de la línea de comandos y variables de entorno

Si ha escrito anteriormente programas de estilo Unix diseñados para ser invocados desde un terminal u otra interfaz de línea de comandos, sabe que estos programas suelen obtener su entrada principalmente de los argumentos de la línea de comandos y, en segundo lugar, de las variables de entorno.

Node sigue estas convenciones de Unix. Un programa Node puede leer sus argumentos de línea de comandos desde el array de cadenas `process.argv`. El primer elemento de esta matriz es siempre la ruta del ejecutable de Node. El segundo argumento es la ruta del archivo de código JavaScript que Node está ejecutando. El resto de los elementos de esta matriz son los argumentos separados por espacios que usted pasó en la línea de comandos cuando invocó a Node.

Por ejemplo, supongamos que guarda este programa Node muy corto en el archivo `argv.js`:

```
console.log(process.argv);
```

A continuación, puede ejecutar el programa y ver una salida como ésta:

```
$ node --trace-uncaught argv.js --arg1 --arg2 filename
[
  '/usr/local/bin/node',
  '/private/tmp(argv.js'),
```

```
'--arg1',
'--arg2',
"nombre de archivo
]
```

Hay un par de cosas que hay que tener en cuenta aquí:

- El primer y segundo elemento de process.argv serán rutas de sistema de archivos completamente cualificadas al ejecutable de Node y al archivo de JavaScript que se está ejecutando, aunque no los hayas escrito así.
- Los argumentos de la línea de comandos que están destinados e interpretados por el propio ejecutable Node son consumidos por el ejecutable Node y no aparecen en process.argv. (El argumento de línea de comandos -trace-uncaught no está haciendo nada útil en el ejemplo anterior; sólo está ahí para demostrar que no aparece en la salida). Cualquier argumento (como --arg1 y filename) que aparezca después del nombre del archivo JavaScript aparecerá en process.argv.

Los programas de Node también pueden tomar información de las variables de entorno al estilo de Unix. Node las hace disponibles a través del objeto process.env. Los nombres de las propiedades de este objeto son nombres de variables de entorno, y los valores de las propiedades (siempre cadenas) son los valores de esas variables.

Esta es una lista parcial de las variables de entorno en mi sistema:

```
$ node -p -e 'process.env'
{
  SHELL: '/bin/bash',
  USUARIO: 'david',
  PATH: '/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin',
  PWD: '/tmp',
  LANG: 'en_US.UTF-8',
```

```
    HOME: '/Users/david',  
}
```

Puede utilizar node -h o node --help para saber qué hacen los argumentos de la línea de comandos -p y -e. Sin embargo, como sugerencia, observe que podría reescribir la línea anterior como node --eval 'process.env' -print.

### 16.1.3 Ciclo de vida del programa

El comando node espera un argumento en la línea de comandos que especifica el archivo de código JavaScript que se va a ejecutar. Este archivo inicial suele importar otros módulos de código JavaScript, y también puede definir sus propias clases y funciones. Fundamentalmente, sin embargo, Node ejecuta el código JavaScript en el archivo especificado de arriba a abajo. Algunos programas Node salen cuando terminan de ejecutar la última línea de código en el archivo. Sin embargo, a menudo un programa Node seguirá ejecutándose mucho después de que se haya ejecutado el archivo inicial. Como discutiremos en las siguientes secciones, los programas Node son a menudo asíncronos y se basan en callbacks y manejadores de eventos. Los programas Node no salen hasta que terminan de ejecutar el archivo inicial y hasta que todos los callbacks han sido llamados y no hay más eventos pendientes. Un programa servidor basado en Node que escucha las conexiones de red entrantes se ejecutará teóricamente para siempre porque siempre estará esperando más eventos.

Un programa puede forzar su salida llamando a process.exit(). Los usuarios normalmente pueden terminar un programa Node

escribiendo Ctrl-C en la ventana de la terminal donde se está ejecutando el programa. Un programa puede ignorar Ctrl-C registrando una función manejadora de señales con `process.on("SIGINT", ()=>{})`.

Si el código de tu programa lanza una excepción y ninguna cláusula `catch` la atrapa, el programa imprimirá un stack trace y saldrá. Debido a la naturaleza asíncrona de Node, las excepciones que se producen en las devoluciones de llamada o en los manejadores de eventos deben ser manejados localmente o no ser manejados en absoluto, lo que significa que el manejo de las excepciones que se producen en las partes asíncronas de su programa puede ser un problema difícil. Si no quieres que estas excepciones provoquen que tu programa se bloquee por completo, registra una función manejadora global que sea invocada en lugar de bloquearse:

```
process.setUncaughtExceptionCaptureCallback(e => { console.  
error("Uncaught exception:", e); });
```

Una situación similar se presenta si una promesa creada por su programa es rechazada y no hay una invocación `.catch()` para manejárla. A partir de Node 13, esto no es un error fatal que hace que su programa salga, pero imprime un mensaje de error verbose en la consola. En alguna versión futura de Node, se espera que los rechazos de promesas no manejados se conviertan en errores fatales. Si no quieres que los rechazos no manejados, impriman mensajes de error o terminen tu programa, registra una función global de manejo:

```
process.on("unhandledRejection", (reason, promise) => {  
  // la razón es que cualquier valor que se hubiera pasado a un
```

```
Función .catch()  
  // la promesa es el objeto Promise que rechazó  
});
```

## 16.1.4 Módulos de nodos

El [capítulo 10](#) documenta los sistemas de módulos de JavaScript, cubriendo tanto

Módulos de Node y módulos de ES6. Como Node fue creado antes de que JavaScript tuviera un sistema de módulos, Node tuvo que crear el suyo propio. El sistema de módulos de Node utiliza la función `require()` para importar valores a un módulo y el objeto `exports` o la propiedad `module.exports` para exportar valores desde un módulo. Estos son una parte fundamental del modelo de programación de Node, y se cubren en detalle en [§10.2](#).

Node 13 añade soporte para los módulos estándar de ES6 así como para los módulos basados en `require` (que Node llama "módulos CommonJS"). Los dos sistemas de módulos no son totalmente compatibles, por lo que esto es algo complicado de hacer. Node necesita saber -antes de cargar un módulo- si ese módulo utilizará `require()` y `module.exports` o si utilizará `import` y `export`. Cuando Node carga un archivo de código JavaScript como un módulo CommonJS, define automáticamente la función `require()` junto con los identificadores `exports` y `module`, y no habilita las palabras clave `import` y `export`. Por otro lado, cuando Node carga un archivo de código como un módulo ES6, debe habilitar las declaraciones `import` y `export`, y *no* debe definir identificadores extra como `require`, `module` y `exports`.

La forma más sencilla de decirle a Node qué tipo de módulo está cargando es codificar esta información en la extensión del archivo. Si guarda su código JavaScript en un archivo que termina con `.mjs`, entonces Node siempre lo cargará como un módulo ES6, esperará que use `import` y `export`, y no proporcionará una función `require()`. Y si guardas tu código en un archivo que termina con `.cjs`, entonces Node siempre lo tratará como un módulo CommonJS, proporcionará una función `require()`, y lanzará un `SyntaxError` si usas declaraciones `import` o `export`.

Para los archivos que no tienen una extensión explícita `.mjs` o `.cjs`, Node busca un archivo llamado `package.json` en el mismo directorio que el archivo y luego en cada uno de los directorios que lo contienen. Una vez encontrado el archivo `package.json` más cercano, Node comprueba si hay una propiedad de tipo de nivel superior en el objeto JSON. Si el valor de la propiedad `type` es `"module"`, entonces Node carga el archivo como un módulo ES6. Si el valor de esa propiedad es `"commonjs"`, entonces Node carga el archivo como un módulo CommonJS. Tenga en cuenta que no es necesario tener un archivo `package.json` para ejecutar programas Node: cuando no se encuentra tal archivo (o cuando se encuentra el archivo pero no tiene una propiedad de tipo), Node utiliza por defecto módulos CommonJS. Este truco de `package.json` sólo es necesario si quieres usar módulos ES6 con Node y no quieres usar la extensión de archivo `.mjs`.

Debido a que hay una enorme cantidad de código Node existente escrito usando el formato de módulo CommonJS, Node permite que los módulos ES6 carguen módulos CommonJS usando la palabra clave `import`. Sin embargo, lo contrario no es cierto: un

módulo CommonJS no puede utilizar require() para cargar un módulo ES6.

### 16.1.5 El gestor de paquetes de nodos

Cuando instalas Node, normalmente obtienes también un programa llamado npm. Este es el gestor de paquetes de Node, y le ayuda a descargar y gestionar las bibliotecas de las que depende su programa. npm mantiene un registro de esas dependencias (así como otra información sobre su programa) en un archivo llamado *package.json* en el directorio raíz de su proyecto. Este archivo *package.json* creado por npm es donde añadirías "type": "module" si quisieras usar módulos ES6 para tu proyecto.

Este capítulo no cubre npm en detalle (pero vea §17.4 para un poco más de profundidad). Lo menciono aquí porque, a menos que escribas programas que no usen ninguna librería externa, es casi seguro que usarás npm o una herramienta parecida.

Supongamos, por ejemplo, que vas a desarrollar un servidor web y piensas utilizar el framework Express (<https://expressjs.com>) para simplificar la tarea. Para empezar, puedes crear un directorio para tu proyecto, y luego, en ese directorio escribe `npm init`. npm te pedirá el nombre de tu proyecto, el número de versión, etc., y luego creará un archivo *package.json* inicial basado en tus respuestas.

Ahora, para empezar a usar Express, debes escribir `npm install express`. Esto le dice a npm que descargue la biblioteca Express junto con todas sus dependencias e instale todos los paquetes en un directorio local *node\_modules/*:

```
$ npm install express
npm notice ha creado un archivo de bloqueo como package-lock.json. Debe confirmar
este archivo.
npm WARN my-server@1.0.0 Sin descripción npm WARN my-
server@1.0.0 Sin campo de repositorio.

+ express@4.17.1
ha añadido 50 paquetes de 37 colaboradores y ha auditado 126 paquetes en
3.058s ha encontrado 0 vulnerabilidades
```

Cuando instalas un paquete con npm, npm registra esta dependencia -que tu proyecto depende de Express- en el archivo package.json. Con esta dependencia registrada en package.json, podrías dar a otro programador una copia de tu código y tu package.json, y podría simplemente escribir npm install para descargar e instalar automáticamente todas las bibliotecas que tu programa necesita para funcionar.

## 16.2 El nodo es asíncrono por defecto

JavaScript es un lenguaje de programación de propósito general, por lo que es perfectamente posible escribir programas de uso intensivo de la CPU que multipliquen grandes matrices o realicen complicados análisis estadísticos. Pero Node fue diseñado y optimizado para programas -como los servidores de red- que hacen un uso intensivo de la E/S. Y en particular, Node fue diseñado para hacer posible la implementación de servidores altamente concurrentes que puedan manejar muchas solicitudes al mismo tiempo.

Sin embargo, a diferencia de muchos lenguajes de programación, Node no logra la concurrencia con hilos. La programación multihilo es notoriamente difícil de hacer correctamente, y difícil de depurar. Además, los hilos son una abstracción relativamente

pesada y si quieras escribir un servidor que pueda manejar cientos de peticiones concurrentes, usando cientos de

hilos puede requerir una cantidad prohibitiva de memoria. Así que Node adopta el modelo de programación de un solo hilo de JavaScript que utiliza la web, y esto resulta ser una gran simplificación que hace que la creación de servidores de red sea

#### **VERDADERO PARALELISMO CON EL NODO**

Los programas Node pueden ejecutar múltiples procesos del sistema operativo, y Node 10 y posteriores soportan objetos Worker (§16.11), que son un tipo de hilo prestado de los navegadores web. Si utiliza múltiples

o crear uno o más hilos de trabajo y ejecutar su programa en un sistema con más de una CPU, entonces su programa ya no será de un solo hilo y su programa realmente estará ejecutando múltiples flujos de código en paralelo. Estas técnicas pueden ser valiosas para las operaciones intensivas de la CPU, pero no se utilizan comúnmente para los programas intensivos de E/S como los servidores.

Sin embargo, cabe destacar que los procesos y trabajadores de Node evitan la complejidad típica de la programación multihilo porque la comunicación entre procesos y trabajadores se realiza mediante el paso de mensajes y no pueden compartir fácilmente la memoria entre ellos.

una habilidad rutinaria en lugar de un arcano.

Node logra altos niveles de concurrencia mientras mantiene un modelo de programación de un solo hilo haciendo que su API sea asíncrona y no bloqueante por defecto. Node se toma su enfoque de no bloqueo muy en serio y hasta un extremo que

puede sorprenderte. Probablemente esperes que las funciones que leen y escriben en la red sean asíncronas, pero Node va más allá y define funciones asíncronas no bloqueantes para leer y escribir archivos del sistema de archivos local. Esto tiene sentido, si lo piensas: la API de Node fue diseñada en los días en que los discos duros giratorios eran todavía la norma y había realmente milisegundos de "tiempo de búsqueda" de bloqueo mientras se esperaba a que el disco girara antes de poder comenzar una operación de archivo. Y en los centros de datos modernos, el sistema de archivos "local" puede estar en realidad a través de la red en algún lugar con latencias de red además de las latencias del disco. Pero incluso si la lectura de un archivo de forma asíncrona le parece normal, Node lo lleva aún más lejos: las funciones por defecto para iniciar una conexión de red o buscar la hora de modificación de un archivo, por ejemplo, también son no bloqueantes.

Algunas funciones de la API de Node son síncronas pero no bloqueantes: se ejecutan hasta su finalización y regresan sin necesidad de bloquearse. Pero la mayoría de las funciones interesantes realizan algún tipo de entrada o salida, y éstas son funciones asíncronas para evitar el más mínimo bloqueo. Node fue creado antes de que JavaScript tuviera una clase Promise, por lo que las APIs asíncronas de Node están basadas en callbacks. (Si aún no has leído o ya has olvidado [el capítulo 13](#), este sería un buen momento para volver a ese capítulo). Generalmente, el último argumento que se pasa a una función asíncrona de Node es un callback. Node utiliza las *devoluciones de llamada* "errorfirst", que normalmente se invocan con dos argumentos. El primer argumento de una llamada de retorno de error es

normalmente nulo en el caso de que no se haya producido ningún error, y el segundo argumento es cualquier dato o respuesta producida por la función asíncrona original que se ha llamado. La razón por la que se pone el argumento de error primero es para que sea imposible omitirlo, y siempre se debe comprobar si este argumento tiene un valor no nulo. Si es un objeto Error, o incluso un código de error entero o un mensaje de error de cadena, entonces algo salió mal. En este caso, es probable que el segundo argumento de su función de devolución de llamada sea nulo.

El siguiente código demuestra cómo utilizar la función no bloqueante readFile() para leer un archivo de configuración, analizarlo como JSON, y luego pasar el objeto de configuración analizado a otra llamada de retorno:

```
const fs = require("fs"); // Requerir el módulo de sistema de archivos // Leer un  
archivo de configuración, analizar su contenido como JSON, y pasar
```

```

el
// valor resultante al callback. Si algo va mal,
// imprimir un mensaje de error en stderr e invocar la llamada de retorno con null
function readConfigFile(path, callback) { fs.readFile(path, "utf8", (err, text) => { if
(err) { // Algo salió mal al leer el archivo
    console.error(err); callback(null); return; }
let data = null; try { data = JSON.parse(text);
} catch(e) { // Algo salió mal al analizar el contenido del archivo console.
error(e); }
callback(data); });
}

```

Node.js es anterior a las promesas estandarizadas, pero dado que es bastante consistente en cuanto a sus devoluciones de llamada por error, es fácil crear variantes basadas en promesas de sus APIs basadas en devoluciones de llamada utilizando la envoltura `util.promisify()`. Así es como podríamos reescribir la función `readConfigFile()` para que devuelva una Promise:

```

const util = require("util"); const fs = require("fs"); // Requiere el módulo de sistema de
archivos const pfs = { // Variantes basadas en promesas de algunas funciones fs
  readFile: util.promisify(fs.readFile) };

function readConfigFile(path) { return pfs.readFile(path, "utf-8").then(text
=> {
  return JSON.parse(text); });
}

```

También podemos simplificar la función anterior basada en promesas utilizando `async` y `await` (de nuevo, si no has leído aún el [capítulo 13](#), este sería un buen momento para hacerlo):

```
async function readConfigFile(path) { let text = await pfs.  
readFile(path, "utf-8"); return JSON.parse(text); }
```

La envoltura `util.promisify()` puede producir una versión basada en promesas de muchas funciones de Node. En Node 10 y posteriores, el objeto `fs.promises` tiene una serie de funciones predefinidas basadas en promesas para trabajar con el sistema de archivos. Las discutiremos más adelante en este capítulo, pero tenga en cuenta que en el código anterior, podríamos reemplazar `pfs.readFile()` con `fs.promises.readFile()`.

Hemos dicho que el modelo de programación de Node es asíncrono por defecto. Pero para la comodidad del programador, Node define variantes bloqueantes y sincrónicas de muchas de sus funciones, especialmente en el módulo del sistema de archivos. Estas funciones suelen tener nombres claramente etiquetados con `Sync` al final.

Cuando un servidor está arrancando por primera vez y está leyendo sus archivos de configuración, todavía no está manejando peticiones de red, y poca o ninguna concurrencia es posible. Así que en esta situación, realmente no hay necesidad de evitar el bloqueo, y podemos utilizar con seguridad funciones de bloqueo como `fs.readFileSync()`. Podemos eliminar el `async` y `await` de este código y escribir una versión puramente sincrónica de nuestra función `readConfigFile()`. En lugar de invocar un

callback o devolver una Promise, esta función simplemente devuelve el valor JSON analizado o lanza una excepción:

```
const fs = require("fs"); function readConfigFileSync(path) { let  
text = fs.readFileSync(path, "utf-8"); return JSON.parse(text); }
```

Además de sus devoluciones de llamada de dos argumentos en caso de error, Node también tiene una serie de APIs que utilizan la asincronía basada en eventos, normalmente para manejar el flujo de datos. Cubriremos los eventos de Node con más detalle más adelante.

Ahora que hemos discutido la agresiva API no bloqueante de Node, volvamos al tema de la concurrencia. Las funciones no bloqueantes incorporadas en Node funcionan utilizando la versión del sistema operativo de las devoluciones de llamada y los manejadores de eventos. Cuando llamas a una de estas funciones, Node actúa para iniciar la operación, y luego registra algún tipo de manejador de eventos con el sistema operativo para que se le notifique cuando la operación esté completa. El callback que pasaste a la función de Node se almacena internamente para que Node pueda invocar su callback cuando el sistema operativo envíe el evento apropiado a Node.

Este tipo de concurrencia suele denominarse concurrencia basada en eventos. En su núcleo, Node tiene un único hilo que ejecuta un "bucle de eventos". Cuando un programa Node se inicia, ejecuta cualquier código que le hayas dicho que ejecute. Este código, presumiblemente, llama al menos a una función no bloqueante que hace que se registre un callback o un manejador de eventos en el sistema operativo. (Si no es así, entonces has

escrito un programa Node síncrono, y Node simplemente sale cuando llega al final). Cuando Node llega al final de su programa, se bloquea hasta que se produce un evento, momento en el que el SO lo vuelve a poner en marcha. Node asigna el evento del SO a la llamada de retorno de JavaScript que registraste y luego invoca esa función. Tu función callback puede invocar más funciones de Node que no se bloquean, haciendo que se registren más manejadores de eventos del SO. Una vez que tu función callback termina de ejecutarse, Node vuelve a dormir y el ciclo se repite.

Para los servidores web y otras aplicaciones de E/S intensivas que pasan la mayor parte de su tiempo esperando la entrada y la salida, este estilo de concurrencia basada en eventos es eficiente y eficaz. Un servidor web puede manejar concurrentemente las solicitudes de 50 clientes diferentes sin necesidad de 50 hilos diferentes, siempre y cuando utilice APIs no bloqueantes y haya algún tipo de mapeo interno de los sockets de red a las funciones de JavaScript para invocar cuando la actividad se produce en esos sockets.

## 16.3 Bufferes

Uno de los tipos de datos que probablemente utilizará con frecuencia en Node -especialmente cuando lea datos de archivos o de la red- es la clase Buffer. Un Buffer es muy parecido a una cadena, excepto que es una secuencia de bytes en lugar de una secuencia de caracteres. Node fue creado antes de que el núcleo de JavaScript soportara arrays tipados (ver §11.2) y no había

Uint8Array para representar una matriz de bytes sin signo. El nodo definió el

Buffer para cubrir esa necesidad. Ahora que Uint8Array es parte de la clase

En el lenguaje JavaScript, la clase Buffer de Node es una subclase de Uint8Array.

Lo que distingue a Buffer de su superclase Uint8Array es que está diseñado para interoperar con cadenas de JavaScript: los bytes de un buffer pueden inicializarse a partir de cadenas de caracteres o convertirse en cadenas de caracteres. Una codificación de caracteres asigna cada carácter de un conjunto de caracteres a un número entero. Dada una cadena de texto y una codificación de caracteres, podemos *codificar* los caracteres de la cadena en una secuencia de bytes. Y dada una secuencia de bytes (correctamente codificada) y una codificación de caracteres, podemos *decodificar* esos bytes en una secuencia de caracteres. La clase Buffer de Node tiene métodos que realizan tanto la codificación como la decodificación, y puedes reconocer estos métodos porque esperan un argumento de codificación que especifica la codificación a utilizar.

Las codificaciones en Node se especifican por nombre, como cadenas. Las codificaciones soportadas son:

"utf8"

Este es el valor por defecto cuando no se especifica ninguna codificación, y es la codificación Unicode que más probablemente utilizará.

"utf16le"

Caracteres Unicode de dos bytes, con ordenación little-endian. Los puntos de código por encima de \uffff se codifican como un par de secuencias de dos bytes. La codificación "ucs2" es un alias.

#### "latin1"

La codificación ISO-8859-1 de un byte por carácter define un conjunto de caracteres adecuado para muchos idiomas de Europa Occidental. Dado que existe un mapeo uno a uno entre los bytes y los caracteres latinos-1, esta codificación también se conoce como "binaria". "ascii"

La codificación ASCII de 7 bits en inglés, un subconjunto estricto de la codificación "utf8".

#### "hex"

Esta codificación convierte cada byte en un par de dígitos hexadecimales ASCII.

#### "base64"

Esta codificación convierte cada secuencia de tres bytes en una secuencia de cuatro caracteres ascii.

Aquí hay un código de ejemplo que demuestra cómo trabajar con Buffers y cómo convertir a y desde cadenas:

```
let b = Buffer.from([0x41, 0x42, 0x43]); // <Buffer 41 42 43>
b.toString() // => "ABC"; por defecto "utf8"
b.toString("hex") // => "414243"

let computer = Buffer.from("IBM3111", "ascii"); // Convertir cadena a Buffer
for(let i = 0; i < computer.length; i++) { // Utiliza el Buffer como matriz de bytes
  computer[i]--;
} // Los buffers son mutables
computer.toString("ascii") // => "HAL2000"
computer.subarray(0,3).map(x=> x+1).toString() // => "IBM"

// Crear nuevos buffers "vacíos" con Buffer.alloc() let zeros = Buffer.alloc(1024); //
1024 ceros let ones = Buffer.alloc(128, 1); // 128 ones let dead = Buffer.alloc(1024,
"DEADBEEF", "hex"); // Patrón de repetición de bytes
```

```
// Los búferes tienen métodos para leer y escribir valores multibyte // desde y hacia  
un búfer en cualquier desplazamiento especificado.  
dead.readUInt32BE(0) // => 0xDEADBEEF dead.readUInt32BE(1) // => 0xADBEEFDE  
dead.readBigUInt64BE(6) // => 0xBEEFDEADBEEFDEADn dead.readUInt32LE(1020)  
// => 0xEFBEADDE
```

Si escribes un programa Node que realmente manipula datos binarios, puedes encontrarte usando la clase Buffer extensamente. Por otro lado, si sólo trabajas con texto que se lee o escribe en un archivo o en la red, entonces sólo encontrarás Buffer como una representación intermedia de tus datos. Varias APIs de Node pueden tomar la entrada o devolver la salida como cadenas u objetos Buffer. Normalmente, si pasas una cadena, o esperas que se devuelva una cadena, desde una de estas APIs, tendrás que especificar el nombre de la codificación de texto que quieras usar. Y si haces esto, entonces puede que no necesites usar un objeto Buffer en absoluto.

## 16.4 Eventos y EventEmitter

Como se ha descrito, todas las APIs de Node son asíncronas por defecto. Para muchas de ellas, esta asincronía adopta la forma de devoluciones de llamada de dos argumentos que se invocan cuando la operación solicitada se ha completado. Pero algunas de las APIs más complicadas se basan en eventos. Este es el caso típico cuando la API está diseñada en torno a un objeto en lugar de una función, o cuando una función de devolución de llamada debe ser invocada varias veces, o cuando hay varios tipos de funciones de devolución de llamada que pueden ser necesarias. Consideremos la clase `net.Server`, por ejemplo: un objeto de este tipo es un socket de servidor que se utiliza para aceptar

conexiones entrantes de clientes. Emite un evento "listening" cuando empieza a escuchar conexiones, un evento "connection" cada vez que un cliente se conecta, y un evento "close" cuando se ha cerrado y ya no está escuchando.

En Node, los objetos que emiten eventos son instancias de EventEmitter o una subclase de EventEmitter:

```
const EventEmitter = require("events"); // El nombre del módulo no coincide con el nombre de la clase const net = require("net"); let server = new net.Server(); // crear un objeto servidor server instanceof EventEmitter // => true: Los servidores son EventEmitters
```

La principal característica de los EventEmitters es que permiten registrar funciones manejadoras de eventos con el método on(). Los EventEmitters pueden emitir múltiples tipos de eventos, y los tipos de eventos se identifican por su nombre. Para registrar un manejador de eventos, llame al método on(), pasando el nombre del tipo de evento y la función que debe ser invocada cuando se produce un evento de ese tipo. Los EventEmitters pueden invocar funciones manejadoras con cualquier número de argumentos, y necesitas leer la documentación para un tipo de evento específico de un EventEmitter específico para saber qué argumentos debes esperar que se pasen:

```
const net = require("net"); let server = new net.Server(); // crear un objeto Server
server.on("connection", socket => { // Escuchar
  "Eventos de conexión"
  // A los eventos de "conexión" del servidor se les pasa un objeto socket
  // para el cliente que acaba de conectarse. Aquí enviamos algunos
  data // al cliente y desconectar. socket.end("Hello
  World", "utf8"); });

```

Si prefiere nombres de métodos más explícitos para registrar escuchadores de eventos, también puede utilizar addListener(). Y puede eliminar un receptor de eventos previamente registrado con off() o removeListener(). Como caso especial, puede registrar un oyente de eventos que se eliminará automáticamente después de que se active por primera vez llamando a once() en lugar de a on().

Cuando se produce un evento de un tipo determinado para un objeto EventEmitter concreto, Node invoca todas las funciones de control que están registradas en ese EventEmitter para eventos de ese tipo. Se invocan en orden desde el primero registrado hasta el último. Si hay más de una función manejadora, se invocan secuencialmente en un solo hilo: no hay paralelismo en Node, recuerda. Y, lo que es más importante, las funciones manejadoras de eventos se invocan de forma sincrónica, no asincrónica. Lo que esto significa es que el método emit() no pone en cola los manejadores de eventos para ser invocados en algún momento posterior. emit() invoca todos los manejadores registrados, uno tras otro, y no regresa hasta que el último manejador de eventos haya regresado.

Lo que esto significa, en efecto, es que cuando una de las APIs incorporadas de Node emite un evento, esa API está básicamente bloqueando sus manejadores de eventos. Si escribes un manejador de eventos que llama a una función de bloqueo como fs.readFileSync(), no se producirá ningún otro manejo de eventos hasta que tu lectura sincrónica de archivos esté completa. Si tu programa es uno -como un servidor de red- que necesita responder, entonces es importante que mantengas tus funciones

manejadoras de eventos no bloqueantes y rápidas. Si necesitas hacer muchos cálculos cuando ocurre un evento, a menudo es mejor usar el manejador para programar ese cálculo asincrónicamente usando `setTimeout()` (ver §11.10). Node también define `setImmediate()`, que programa una función para ser invocada inmediatamente después de que todas las devoluciones de llamada y eventos pendientes hayan sido manejados.

La clase `EventEmitter` también define un método `emit()` que hace que se invoquen las funciones del manejador de eventos registrado. Esto es útil si estás definiendo tu propia API basada en eventos, pero no es comúnmente usado cuando sólo estás programando con APIs existentes. `emit()` debe ser invocado con el nombre del tipo de evento como su primer argumento. Cualquier argumento adicional que se pase a `emit()` se convierte en argumento de las funciones manejadoras de eventos registradas. Las funciones manejadoras también son invocadas con el valor `this` establecido en el propio objeto `EventEmitter`, lo cual es a menudo conveniente. (Recuerde, sin embargo, que las funciones de flecha siempre utilizan el valor `this` del contexto en el que están definidas, y no pueden ser invocadas con ningún otro valor `this`. No obstante, las funciones de flecha son a menudo la forma más conveniente de escribir manejadores de eventos).

Cualquier valor devuelto por una función manejadora de eventos es ignorado. Sin embargo, si una función manejadora de eventos lanza una excepción, ésta se propaga desde la llamada a `emit()` e impide la ejecución de cualquier función manejadora que haya sido registrada después de la que lanzó la excepción.

Recuerda que las APIs basadas en callbacks de Node utilizan callbacks de error primero, y es importante que siempre compruebes el primer argumento del callback para ver si se ha producido un error. Con las APIs basadas en eventos, el equivalente son los eventos de "error". Dado que las APIs basadas en eventos se utilizan a menudo para la creación de redes y otras formas de flujo de E/S, son vulnerables a errores asíncronos impredecibles, y la mayoría de los EventEmitters definen un evento "error" que emiten cuando se produce un error. Siempre que utilices una API basada en eventos, deberías tener la costumbre de registrar un manejador para los eventos "error". Los eventos "error" reciben un tratamiento especial por parte de la clase EventEmitter. Si se llama a emit() para emitir un evento "error", y si no hay manejadores registrados para ese tipo de evento, entonces se lanzará una excepción. Como esto ocurre de forma asíncrona, no hay forma de manejar la excepción en un bloque catch, por lo que este tipo de error suele provocar la salida del programa.

## 16.5 Arroyos

Cuando se implementa un algoritmo para procesar datos, casi siempre es más fácil leer todos los datos en la memoria, hacer el procesamiento y luego escribir los datos. Por ejemplo, podrías escribir una función de Nodo para copiar un archivo de esta manera.<sup>1</sup>

```
const fs = require("fs");

// Una función asíncrona pero no de flujo (y por lo tanto ineficiente).
function copyFile(sourceFilename, destinationFilename, callback) { fs.
readFile(sourceFilename, (err, buffer) => { if (err) { callback(err); } else {
```

```
    fs.writeFile(destinationFilename, buffer, callback);

  }
});

}
```

Esta función `copyFile()` utiliza funciones asíncronas y callbacks, por lo que no se bloquea y es adecuada para su uso en programas concurrentes como los servidores. Pero observe que debe asignar suficiente memoria para mantener todo el contenido del archivo en memoria a la vez. Esto puede estar bien en algunos casos de uso, pero empieza a fallar si los archivos a copiar son muy grandes, o si su programa es altamente concurrente y puede haber muchos archivos siendo copiados al mismo tiempo. Otro defecto de esta implementación de `copyFile()` es que no puede comenzar a escribir el nuevo archivo hasta que haya terminado de leer el archivo antiguo.

La solución a estos problemas es utilizar algoritmos de flujo en los que los datos "fluyen" hacia el programa, se procesan y luego salen del mismo. La idea es que tu algoritmo procese los datos en pequeños trozos y que el conjunto de datos completo nunca se mantenga en la memoria a la vez. Cuando las soluciones de streaming son posibles, son más eficientes en cuanto a la memoria y también pueden ser más rápidas. Las APIs de red de Node están basadas en streaming y el módulo de sistema de archivos de Node define APIs de streaming para la lectura y escritura de archivos, por lo que es probable que utilices una API de streaming en muchos de los programas Node que escribas. Veremos una versión de streaming de la función `copyFile()` en "modo de flujo".

Node admite cuatro tipos básicos de flujo:

### *Legible*

Los flujos legibles son fuentes de datos. El flujo devuelto por `fs.createReadStream()`, por ejemplo, es un flujo del que se puede leer el contenido de un archivo especificado.

`process.stdin` es otro flujo legible que devuelve datos de la entrada estándar.

### *Se puede escribir en*

Los flujos de escritura son sumideros o destinos de datos. El valor de retorno de `fs.createWriteStream()`, por ejemplo, es un Writable stream: permite que los datos se escriban en él en trozos, y da salida a todos esos datos en un archivo especificado.

### *Dúplex*

Los flujos dúplex combinan un flujo de lectura y un flujo de escritura en un solo objeto. Los objetos Socket devueltos por `net.connect()` y otras APIs de red de Node, por ejemplo, son flujos dúplex. Si escribe en un socket, sus datos se envían a través de la red a cualquier ordenador al que esté conectado el socket. Y si lees desde un socket, accedes a los datos escritos por ese otro ordenador.

### *Transformar*

Los flujos de transformación también se pueden leer y escribir, pero difieren de los flujos dúplex en un aspecto importante: los datos que se escriben en un flujo de transformación se pueden leer, normalmente en alguna forma transformada, desde el mismo flujo. La función `zlib.createGzip()` por ejemplo, devuelve un flujo Transform que comprime (con el algoritmo `gzip`) los datos escritos en él. De manera similar, la función `crypto.createCipheriv()` devuelve un flujo Transform que encripta o desencripta los datos que se escriben en él.

Por defecto, los flujos leen y escriben buffers. Si llamas al método `setEncoding()` de un stream Readable, te devolverá cadenas decodificadas en lugar de objetos Buffer. Y si escribes una cadena en un búfer Writable, se codificará automáticamente utilizando la codificación por defecto del búfer o cualquier codificación que especifiques. La API de flujos de Node también soporta un "modo objeto" en el que los flujos leen y escriben objetos más complejos que los buffers y las cadenas. Ninguna de las APIs del núcleo de Node utiliza este modo objeto, pero puede encontrarlo en otras bibliotecas.

Los flujos legibles tienen que leer sus datos desde algún lugar, y los flujos escribibles tienen que escribir sus datos en algún lugar, por lo que cada flujo tiene dos extremos: una entrada y una salida o una fuente y un destino. Lo complicado de las APIs basadas en flujos es que los dos extremos del flujo casi siempre fluyen a diferentes velocidades. Quizás el código que lee de un flujo quiere leer y procesar los datos más rápidamente que los datos que se escriben en el flujo. O a la inversa: quizás los datos se escriben en un flujo más rápido de lo que se pueden leer y sacar del flujo en el otro extremo. Las implementaciones de flujos casi siempre incluyen un búfer interno para mantener los datos que se han escrito pero aún no se han leído. El almacenamiento en búfer ayuda a asegurar que hay datos disponibles para leer cuando se solicitan, y que hay espacio para mantener los datos cuando se escriben. Pero ninguna de estas cosas puede ser garantizada, y es la naturaleza de la programación basada en flujos que los lectores a veces tendrán que esperar a que los datos sean escritos (porque el buffer del flujo está vacío), y los

escritores a veces tendrán que esperar a que los datos sean leídos (porque el buffer del flujo está lleno).

En los entornos de programación que utilizan la concurrencia basada en hilos, las APIs de flujos suelen tener llamadas de bloqueo: una llamada a la lectura de datos no vuelve hasta que los datos llegan al flujo y una llamada a la escritura de datos se bloquea hasta que hay suficiente espacio en el buffer interno del flujo para acomodar los nuevos datos. Sin embargo, con un modelo de concurrencia basado en eventos, las llamadas de bloqueo no tienen sentido, y las APIs de flujo de Node están basadas en eventos y callbacks. A diferencia de otras APIs de Node, no hay versiones "Sync" de los métodos que se describirán más adelante en este capítulo.

La necesidad de coordinar la legibilidad del flujo (búfer no vacío) y la escritura (búfer no lleno) a través de eventos hace que las APIs de flujo de Node sean algo complicadas. Esto se ve agravado por el hecho de que estas APIs han evolucionado y cambiado a lo largo de los años: para los flujos legibles, hay dos APIs completamente distintas que puedes utilizar. A pesar de la complejidad, vale la pena entender y dominar las APIs de streaming de Node porque permiten una E/S de alto rendimiento en tus programas.

Las subsecciones que siguen demuestran cómo leer y escribir desde las clases de flujo de Node.

### 16.5.1 Tuberías

A veces, necesitas leer datos de un flujo simplemente para dar la vuelta y escribir esos mismos datos en otro flujo. Imagina, por

ejemplo, que estás escribiendo un simple servidor HTTP que sirve un directorio de archivos estáticos. En este caso, necesitarás leer datos de un flujo de entrada de archivos y escribirlos en un socket de red. Pero en lugar de escribir tu propio código para manejar la lectura y la escritura, puedes simplemente conectar los dos sockets juntos como una "tubería" y dejar que Node maneje las complejidades por ti. Simplemente pasa el flujo Writable al método pipe() del flujo Readable:

```
const fs = require("fs");
function pipeFileToSocket(filename, socket) { fs.
createReadStream(filename). pipe(socket); }
```

La siguiente función de utilidad canaliza un flujo a otro e invoca una llamada de retorno cuando termina o cuando se produce un error:

```
function pipe(readable, writable, callback) { // En primer lugar,
configura la gestión de errores function handleError(err) {
readable. close(); writable. close(); callback(err); }

// A continuación define la tubería y maneja el caso de terminación normal legible .
on("error", handleError)
    . pipa(escribible)
    . on("error", handleError)
    . on("finish", callback);
}
```

Los flujos de transformación son particularmente útiles con las tuberías, y crean tuberías que implican más de dos flujos. Aquí hay una función de ejemplo que comprime un archivo:

```

const fs = require("fs"); const zlib =
require("zlib");

function gzip(filename, callback) { // Crear los flujos let source = fs.
createReadStream(filename); let destination = fs.createWriteStream(filename + ".gz");
let gzipper = zlib.createGzip();

    // Configurar el origen de la tubería .on("error", callback) // llamar a la
    devolución de llamada en caso de error de lectura .pipe(gzipper)
    .pipe(destino)
    .on("error", callback) // llama a la devolución de llamada en caso de error
    de escritura .on("finish", callback); // llama a la devolución de llamada cuando
    la escritura ha terminado }

```

Usar el método pipe() para copiar datos de un flujo legible a un flujo escribible es fácil, pero en la práctica, a menudo necesitas procesar los datos de alguna manera mientras fluyen a través de tu programa. Una manera de hacer esto es implementar su propio flujo Transform para hacer ese procesamiento, y este enfoque le permite evitar la lectura y escritura manual de los flujos. Aquí, por ejemplo, hay una función que funciona como la utilidad grep de Unix: lee líneas de texto de un flujo de entrada, pero escribe sólo las líneas que coinciden con una expresión regular especificada:

```
const stream = require("stream");

class GrepStream extends stream.Transform { constructor(pattern) {
super({decodeStrings: false}); // No convertir las cadenas en buffers this. pattern =
pattern; // La expresión regular que queremos que coincida con this. incompleteLine =
""; // Cualquier resto del último trozo de datos }

// Este método se invoca cuando hay una cadena lista para ser
// transformado. Debe pasar los datos transformados al
// función de devolución de llamada. Esperamos una entrada de cadena, por lo que
esta
```

```
stream sólo debe // conectarse a streams legibles a los que se haya // llamado a
setEncoding(). _transform(chunk, encoding, callback) { if (typeof chunk !==
"string") {
    callback(new Error("Esperaba una cadena pero obtuvo un buffer")); return;
}
// Añadir el chunk a cualquier línea previamente incompleta y romper // todo
en líneas let lines = (this. incompleteLine + chunk). split("\n");

// El último elemento de la matriz es la nueva línea incompleta this.
incompleteLine = lines. pop();

// Buscar todas las líneas coincidentes
let output = lines // Comienza con todas las líneas completas, . filter(l => this. pattern.
test(l)) // filtrarlas en busca de coincidencias, . join("\n"); // y volver a unirlas.

// Si algo coincide, añade una nueva línea final if (output) { output +=
"\n"; }

// Llama siempre a la devolución de llamada aunque no haya salida
callback(null, output); }

// Se llama justo antes de cerrar el flujo.
// Es nuestra oportunidad de escribir los últimos datos.
_flush(callback) {
    // Si todavía tenemos una línea incompleta, y coincide // pásala a la
    llamada de retorno if (this. pattern. test(this. incompleteLine)) {
```

```

        callback(null, this.incompleteLine + "\n"); }
    }
}

// Ahora podemos escribir un programa como 'grep' con esta clase. let pattern = new
RegExp(process.argv[2]); // Obtener un RegExp desde la línea de comandos.
procesar.stdin // Empezar con la entrada estándar, .setEncoding("utf8") //
leerlo como
Cadenas Unicode,
.pipe(new GrepStream(pattern)) // lo canaliza a nuestro
GrepStream, .pipe(process.stdout) // y lo canaliza a la salida estándar. .on("error", () => process.exit()); // Salir con gracia si stdout se cierra.

```

## 16.5.2 Iteración asíncrona

En Node 12 y posteriores, los flujos legibles son iteradores asíncronos, lo que significa que dentro de una función asíncrona se puede utilizar un bucle for/await para leer trozos de cadena o de Buffer de un flujo utilizando un código que está estructurado como lo estaría el código síncrono. (Ver [§13.4](#) para más información sobre iteradores asíncronos y bucles for/await).

Usar un iterador asíncrono es casi tan fácil como usar el método pipe(), y es probablemente más fácil cuando necesitas procesar cada trozo que lees de alguna manera. Así es como podríamos reescribir el programa grep de la sección anterior utilizando una función asíncrona y un bucle for/await:

```
// Leer líneas de texto del flujo de origen, y escribir cualquier línea
// que coincidan con el patrón especificado al destino
```

*corriente.*

```
async function grep(source, destination, pattern, encoding="utf8") {  
    // Configurar el flujo de origen para leer cadenas, no  
    Fuente de búferes. setEncoding(encoding);  
  
    // Establecer un controlador de errores en el flujo de destino en caso de que el estándar  
    // la salida se cierra inesperadamente (cuando se canaliza la salida a  
    'head', por ejemplo) destination.on("error", err => process.exit());  
  
    // Es poco probable que los trozos que leemos terminen con una nueva línea, por lo que  
    // cada uno de ellos  
    // probablemente tenga una línea parcial al final. Rastrea eso aquí let  
    incompleteLine = "";  
  
    // Utiliza un bucle for/await para leer asíncronamente trozos del flujo de  
    // entrada for await (let chunk of source) {  
        // Dividir el final del último chunk más éste en líneas let lines = (incompleteLine +  
        chunk).split("\n"); // La última línea está incompleta incompleteLine = lines.pop();  
        // Ahora recorre las líneas y escribe cualquier coincidencia en el destino for(let  
        line of lines) { if (pattern.test(line)) { destination.write(line + "\n", encoding); }  
        }  
        // Por último, comprueba si hay una coincidencia en el texto final. if (pattern.  
        test(incompleteLine)) { destination.write(incompleteLine + "\n", encoding); } }  
  
let pattern = new RegExp(process.argv[2]); // Obtener un RegExp de la línea de  
comandos.
```

```
grep(process.stdin, process.stdout, pattern) // Llama a la función asíncrona  
grep(). .catch(err => { // Maneja las excepciones asíncronas. console.  
error(err); process.exit();});
```

### 16.5.3 Escribir en flujos y manejar la contrapresión

La función `async grep()` en el ejemplo de código anterior demostró cómo usar un flujo `Readable` como un iterador asíncrono, pero también demostró que puedes escribir datos en un flujo `Writable` simplemente pasándolo al método `write()`. El método `write()` toma un buffer o una cadena como primer argumento. (Los flujos de objetos esperan otros tipos de objetos, pero están fuera del alcance de este capítulo). Si pasas un buffer, los bytes de ese buffer se escribirán directamente. Si pasas una cadena, será codificada en un buffer de bytes antes de ser escrita. Los flujos escribibles tienen una codificación por defecto que se utiliza cuando se pasa una cadena como único argumento a `write()`. La codificación por defecto es típicamente "utf8", pero puedes establecerla explícitamente llamando a `setDefaultEncoding()` en el flujo `Writable`. Alternativamente, cuando se pasa una cadena como primer argumento a `write()` se puede pasar un nombre de codificación como segundo argumento.

`write()` toma opcionalmente una función de devolución de llamada como su tercer argumento. Esta será invocada cuando los datos hayan sido realmente escritos y ya no estén en el buffer interno del flujo de escritura. (Esta llamada de retorno también puede ser invocada si se produce un error, pero esto no está garantizado. Deberías registrar un manejador de eventos "error" en el flujo `Writable` para detectar errores).

El método `write()` tiene un valor de retorno muy importante. Cuando se llama a `write()` en un flujo, siempre aceptará y almacenará en el búfer el trozo de datos que se ha pasado.

Entonces devuelve true si el buffer interno aún no está lleno. O, si el buffer está lleno o sobrecargado, devuelve false. Este valor de retorno es consultivo, y puede ignorarlo: los flujos escribibles ampliarán su búfer interno tanto como sea necesario si sigue llamando a write(). Pero recuerde que la razón para usar una API de flujo en primer lugar es evitar el coste de mantener muchos datos en memoria a la vez.

Un valor de retorno de false del método write() es una forma de *contrapresión*: un mensaje del flujo de que has escrito datos más rápido de lo que puede ser manejado. La respuesta adecuada a este tipo de contrapresión es dejar de llamar a write() hasta que el flujo emita un evento de "drenaje", señalando que hay de nuevo espacio en el buffer. Aquí, por ejemplo, hay una función que escribe en un flujo, y luego invoca un callback cuando está bien escribir más datos en el flujo:

```
function write(stream, chunk, callback) { // Escribir el chunk especificado en el stream especificado
  let hasMoreRoom = stream.write(chunk);

  // Comprueba el valor de retorno del método write():
  if (hasMoreRoom) { // Si devuelve true, entonces setImmediate(callback); // invoca el callback de forma asíncrona. } else { // Si devuelve false, entonces stream.once("drain", callback); // invocar callback en evento drain.

}

}
```

El hecho de que a veces esté bien llamar a write() varias veces seguidas y que a veces haya que esperar un evento entre escrituras hace que los algoritmos sean incómodos. Esta es una

de las razones por las que usar el método pipe() es tan atractivo: cuando usas pipe(), Node maneja la contrapresión por ti automáticamente.

Si estás usando await y async en tu programa, y estás tratando flujos legibles como iteradores asíncronos, es sencillo implementar una versión basada en Promise de la función de utilidad write() de arriba para manejar adecuadamente la contrapresión. En la función asíncrona grep() que acabamos de ver, no manejamos la contrapresión. La función async copy() del siguiente ejemplo demuestra cómo se puede hacer correctamente. Observa que esta función sólo copia trozos de un flujo de origen a un flujo de destino y llamar a copy(source, destination) es muy parecido a llamar a source.pipe(destination):

```
// Esta función escribe el chunk especificado en el stream especificado y // devuelve una Promise que se cumplirá cuando esté bien escribir de nuevo. // Como devuelve una Promise, puede usarse con await. function write(stream, chunk) { // Escribe el chunk especificado en el stream especificado let hasMoreRoom = stream.write(chunk);

if (hasMoreRoom) { // Si el buffer no está lleno, devuelve Promise. resolve(null);
// un objeto Promise ya resuelto
} si no {
```

```

    return new Promise(resolve => { // De lo contrario, devuelve una Promise que
      stream.once("drain", resolve); // resuelve en el evento drain. });
  }

  // Copiar los datos del flujo de origen al flujo de destino // respetando la contrapresión
  // del flujo de destino.
  // Esto es muy parecido a llamar a source.pipe(destination). async function
  copy(source, destination) {
    // Establecer un controlador de errores en el flujo de destino en caso de que el estándar
    // la salida se cierra inesperadamente (cuando se canaliza la salida a
    'head', por ejemplo) destination.on("error", err => process.exit());
    // Utiliza un bucle for/await para leer asíncronamente trozos del flujo de
    entrada for await (let chunk of source) {
      // Escribe el chunk y espera hasta que haya más espacio en el buffer.
      await write(destination, chunk);
    }

    // Copiar la entrada estándar a la salida estándar copy(process.stdin, process.
    stdout);
  }
}

```

Antes de que concluyamos esta discusión sobre la escritura en flujos, ten en cuenta de nuevo que no responder a la contrapresión puede hacer que tu programa utilice más memoria de la que debería cuando el buffer interno de un flujo escribible se desborde y crezca cada vez más. Si estás escribiendo un servidor de red, esto puede ser un problema de seguridad remotamente explotable. Supongamos que escribes un servidor HTTP que entrega archivos a través de la red, pero no has utilizado pipe() y no te has tomado el tiempo de manejar la contrapresión del método write(). Un atacante podría escribir un cliente HTTP que inicie solicitudes de archivos grandes (como

imágenes) pero que nunca lea el cuerpo de la solicitud. Dado que el cliente no está leyendo los datos a través de la red, y el servidor no está respondiendo a la contrapresión, los búferes en el servidor se van a desbordar. Con suficientes conexiones concurrentes del atacante, esto puede convertirse en un ataque de denegación de servicio que ralentiza su servidor o incluso lo bloquea.

#### **16.5.4 Lectura de flujos con eventos**

Los flujos de lectura de Node tienen dos modos, cada uno de los cuales tiene su propia API para la lectura. Si no puedes usar tuberías o iteración asíncrona en tu programa, tendrás que elegir una de estas dos APIs basadas en eventos para manejar los flujos. Es importante que utilices sólo una u otra y no mezcles las dos APIs.

#### **MODO DE FLUJO**

En *el modo de flujo*, cuando llegan datos legibles, se emiten inmediatamente en forma de un evento "data". Para leer de un flujo en este modo, simplemente registre un manejador de eventos para los eventos "datos", y el flujo le enviará trozos de datos (buffers o cadenas) tan pronto como estén disponibles. Tenga en cuenta que no es necesario llamar al método `read()` en el modo de flujo: sólo necesita manejar los eventos "data". Tenga en cuenta que los flujos recién creados no comienzan en modo fluido. El registro de un manejador de eventos "data" cambia un flujo al modo de flujo. Convenientemente, esto significa que un flujo no emite eventos de "datos" hasta que se registra el primer manejador de eventos de "datos".

Si está utilizando el modo de flujo para leer datos de un flujo Legible, procesarlos, y luego escribirlos en un flujo Legible, entonces puede necesitar manejar la contrapresión del flujo Legible. Si el método `write()` devuelve `false` para indicar que el buffer de escritura está lleno, puedes llamar a `pause()` en el flujo Legible para detener temporalmente los eventos de datos. Entonces, cuando obtengas un evento de "drenaje" del flujo de escritura, puedes llamar a `resume()` en el flujo de lectura para que los eventos de "datos" vuelvan a fluir.

Un flujo en modo fluido emite un evento "fin" cuando se alcanza el final del flujo. Este evento indica que no se emitirán más eventos "datos". Y, como en todos los flujos, se emite un evento "error" si se produce un error.

Al principio de esta sección sobre flujos, mostramos una función `copyFile()` sin flujo y prometimos que habría una versión mejor. El siguiente código muestra cómo implementar una función `copyFile()` de flujo que utiliza la API de modo de flujo y maneja la contrapresión. Esto habría sido más fácil de implementar con una llamada a `pipe()`, pero sirve aquí como una demostración útil de los múltiples manejadores de eventos que se utilizan para coordinar el flujo de datos de un flujo a otro.

```
const fs = require("fs");

// Una función de copia de archivos en flujo, utilizando el "modo de flujo".
// Copia el contenido del archivo de origen nombrado al archivo de destino nombrado.
// En caso de éxito, invoca la llamada de retorno con un argumento nulo. En caso de
// error, // invoca la llamada de retorno con un objeto Error.
function copyFile(sourceFilename, destinationFilename, callback) { let input = fs.
createReadStream(sourceFilename); let output = fs.
createWriteStream(destinationFilename);
```

```
    input.on("data", (chunk) => { // Cuando obtenemos nuevos datos, let hasRoom =
      output.write(chunk); // escríbelo en el flujo de salida. if (! hasRoom) { // Si el flujo de
      salida está lleno input.pause(); // entonces pausa el flujo de entrada. } });
    input.on("end", () => { // Cuando lleguemos al final de la entrada, output.end(); // indica al
      flujo de salida que termine. });
    input.on("error", err => { // Si obtenemos un error en la
      entrada, callback(err); // llama al callback con el error process.exit(); // y sale. });
  }

  output.on("drain", () => { // Cuando la salida ya no está llena, input.resume(); // reanuda los eventos de datos en la entrada });
  output.on("error", err => { // Si
    obtenemos un error en la salida, callback(err); // llama al callback con el error process.exit(); // y sale. });
  output.on("finish", () => { // Cuando la salida está completamente escrita
    callback(null); // llama al callback sin error. });

// Esta es una sencilla utilidad de línea de comandos para copiar archivos let
from = process.argv[2], to = process.argv[3];

console.log(`Copiando archivo ${de} a ${a}`);
copyFile(from, to, err => {
  if (err) {
    console.error(err);
  } else {
    console.log("hecho.");
  }
});
```

## MODO PAUSA

El otro modo para los flujos legibles es el "modo pausado". Este es el modo en el que los flujos comienzan. Si nunca registra un manejador de eventos "data" y nunca llama al método pipe(), entonces un flujo Legible permanece en modo pausado. En el modo de pausa, el flujo no le envía datos en forma de eventos "data". En su lugar, usted extrae datos del flujo llamando explícitamente a su método read(). Esta no es una llamada de bloqueo, y si no hay datos disponibles para leer en el flujo, devolverá null.

Como no hay una API sincrónica para esperar los datos, el modo pausado

La API también se basa en los eventos. Un flujo legible en modo pausado emite

"readable" cuando los datos están disponibles para leer en el flujo. En respuesta, tu código debe llamar al método read() para leer esos datos. Debes hacer esto en un bucle, llamando a read() repetidamente hasta que devuelva null. Es necesario vaciar completamente el búfer del flujo de esta manera para desencadenar un nuevo evento "legible" en el futuro. Si dejas de llamar a read() mientras todavía hay datos legibles, no obtendrás otro evento "legible" y es probable que tu programa se cuelgue.

Los flujos en modo pausado emiten eventos de "fin" y "error" al igual que los flujos en modo fluido. Si estás escribiendo un programa que lee datos de un flujo legible y los escribe en un flujo escribible, entonces el modo pausado puede no ser una buena opción. Para manejar adecuadamente la contrapresión, sólo quieres leer cuando el flujo de entrada es legible y el flujo de salida no está respaldado. En el modo pausado, eso significa leer y escribir hasta que read() devuelva null o write() devuelva false, y entonces empezar a leer o



escribir de nuevo en un evento de lectura o drenaje. Esto es poco elegante, y usted puede encontrar que el modo de flujo (o tuberías) es más fácil en este caso.

El siguiente código demuestra cómo se puede calcular un hash SHA256 para el contenido de un archivo especificado. Utiliza un flujo legible en modo pausado para leer el contenido de un archivo en trozos, y luego pasa cada trozo al objeto que calcula el hash. (Tenga en cuenta que en Node 12 y posteriores, sería más sencillo escribir esta función utilizando un bucle for/await).

```
const fs = require("fs"); const crypto =
require("crypto");

// Calcula un hash sha256 del contenido del archivo nombrado y pasa el // hash
// (como una cadena) a la función de devolución de llamada del primer error
// especificado.
function sha256(filename, callback) { let input = fs.createReadStream(filename); //
// El flujo de datos.
let hasher = crypto.createHash("sha256"); // Para calcular el hash.

input.on("readable", () => { // Cuando hay datos listos para leer let chunk; while(chunk =
input.read()) { // Lee un chunk, y si no es nulo, hasher.update(chunk); // lo pasa al
hasher,
```

```

        }           // y sigue el bucle hasta que no se pueda leer}); input.on("end",
() => { // Al final del flujo, let hash = hasher.digest("hex"); // calcula el hash,
callback(null, hash); // y lo pasa al callback. });
}); input.on("error", callback); // En caso de error, llama al callback }

// Esta es una sencilla utilidad de línea de comandos para calcular el hash de un
archivo sha256(process.argv[2], (err, hash) => { // Pasar el nombre del archivo desde
la línea de comandos.

  if (err) { // Si obtenemos un error console.error(err.toString()); // imprimirlo como
un error. } else { // En caso contrario, console.log(hash); // imprime la cadena del
hash. }
});

```

## 16.6 Detalles del proceso, la CPU y el sistema operativo

El objeto global Process tiene una serie de propiedades y funciones útiles que generalmente se relacionan con el estado del proceso Node que se está ejecutando actualmente. Consulte la documentación de Node para obtener detalles completos, pero aquí hay algunas propiedades y funciones que debe conocer:

`process.argv` // Una matriz de argumentos de la línea de comandos.

`process.arch` // La arquitectura de la CPU: "x64", para

*ejemplo.*

`process.cwd() // Devuelve el directorio de trabajo actual.`  
`process.chdir() // Establece el directorio de trabajo actual. process.`  
`cpuUsage() // Informa del uso de la CPU.`  
`proceso.env // Un objeto de variables de entorno.`  
`process.execPath // La ruta absoluta del sistema de archivos al ejecutable del nodo. process.exit() // Termina el programa.`  
`process.exitCode // Un código entero para ser reportado cuando el programa se cierra.`  
`process.getuid() // Devuelve el id de usuario Unix del usuario actual.`  
`process.hrtime.bigint() // Devuelve una marca de tiempo en nanosegundos de "alta resolución".`  
`process.kill() // Envía una señal a otro proceso. process.memoryUsage() // Devuelve un objeto con detalles de uso de memoria.`  
`process.nextTick() // Al igual que setImmediate(), invoca una función pronto.`  
`process.pid // El id del proceso actual. process.ppid // El id del proceso padre.`  
`process.platform // El sistema operativo: "linux", "darwin", o "win32", por ejemplo.`  
`process.resourceUsage() // Devuelve un objeto con detalles de uso de recursos.`  
`process.setuid() // Establece el usuario actual, por id o por nombre.`  
`process.title // El nombre del proceso que aparece en los listados `ps`.`  
`process.umask() // Establece o devuelve los permisos por defecto para los nuevos archivos.`  
`process.uptime() // Devuelve el tiempo de actividad del Nodo en segundos. process.version // La cadena de versión del Nodo.`  
`process.versions // Cadenas de versiones de las bibliotecas de las que depende Node.`

El módulo "os" (que, a diferencia de process, necesita ser cargado explícitamente con require()) proporciona acceso a detalles de bajo nivel similares sobre el ordenador y el sistema operativo en el que se está ejecutando Node. Puede que nunca necesites usar

ninguna de estas características, pero vale la pena saber que Node las pone a tu disposición:

```
const os = require("os"); os.arch() // Devuelve la arquitectura de la CPU. "x64" o  
"arm", por ejemplo. os.constants // Constantes útiles como  
os.constants.signals.SIGINT.  
os.cpus() // Datos sobre los núcleos de la CPU del sistema, incluyendo los tiempos de  
uso.  
os.endianness() // El endianness nativo de la CPU "BE" o "LE". os.EOL // El terminador  
de línea nativo del SO: "\n" o "\r\n".  
os.freemem() // Devuelve la cantidad de RAM libre en bytes.  
os.getPriority() // Devuelve la prioridad de programación del SO de un proceso.  
os.homedir() // Devuelve el directorio raíz del usuario actual.  
os.hostname() // Devuelve el nombre del ordenador.  
os.loadavg() // Devuelve las medias de carga de 1, 5 y 15 minutos.  
os.networkInterfaces() // Devuelve detalles sobre las conexiones de red disponibles.  
os.platform() // Devuelve el sistema operativo: "linux", "darwin", o "win32", por  
ejemplo.  
os.release() // Devuelve el número de versión del SO. os.setPriority() // Intenta  
establecer la prioridad de programación de un proceso.  
os.tmpdir() // Devuelve el directorio temporal por defecto.  
os.totalmem() // Devuelve la cantidad total de RAM en bytes.  
os.type() // Devuelve el sistema operativo: "Linux", "Darwin", o "Windows_NT", por  
ejemplo.
```

```
os.uptime() // Devuelve el tiempo de actividad del sistema en segundos.  
os.userInfo() // Devuelve el uid, nombre de usuario, home y shell del usuario actual.
```

## 16.7 Trabajar con archivos

El módulo "fs" de Node es una completa API para trabajar con archivos y directorios. Se complementa con el módulo "path", que define funciones de utilidad para trabajar con nombres de archivos y directorios. El módulo "fs" contiene un puñado de funciones de alto nivel para leer, escribir y copiar archivos fácilmente. Pero la mayoría de las funciones del módulo son enlaces de JavaScript de bajo nivel a las llamadas del sistema Unix (y sus equivalentes en Windows). Si ha trabajado antes con llamadas de bajo nivel al sistema de archivos (en C u otros lenguajes), la API de Node le resultará familiar. Si no es así, puede que algunas partes de la API "fs" le resulten escuetas y poco intuitivas. La función para borrar un archivo, por ejemplo, se llama `unlink()`.

El módulo "fs" define una gran API, principalmente porque suele haber múltiples variantes de cada operación fundamental. Como se discutió al principio del capítulo, la mayoría de las funciones como `fs.readFile()` son no bloqueantes, basadas en callbacks y asíncronas. Sin embargo, normalmente cada una de estas funciones tiene una variante de bloqueo sincrónico, como `fs.readFileSync()`. En Node 10 y posteriores, muchas de estas funciones también tienen una variante asíncrona basada en promesas, como `fs.promises.readFile()`. La mayoría de las funciones "fs" toman una cadena como primer argumento, especificando la ruta (nombre de archivo más los nombres de los directorios opcionales) al archivo que se va a operar. Pero algunas de estas funciones también admiten una variante que toma un "descriptor de archivo" entero como primer argumento

en lugar de una ruta. Estas variantes tienen nombres que comienzan con la letra "f". Por ejemplo, `fs.truncate()` trunca un archivo especificado por la ruta, y `fs.ftruncate()` trunca un archivo especificado por el descriptor de archivo. Hay una `fs.promises.truncate()` basada en promesas que espera una ruta y otra

Versión basada en promesas que se implementa como un método de un objeto `FileHandle`. (La clase `FileHandle` es el equivalente a un descriptor de archivo en la API basada en promesas). Finalmente, hay un puñado de funciones en el módulo "fs" que tienen variantes cuyos nombres llevan el prefijo "l". Estas variantes "l" son como la función base, pero no siguen los enlaces simbólicos en el sistema de archivos, sino que operan directamente sobre los propios enlaces simbólicos.

### 16.7.1 Rutas, descriptores de archivos y `FileHandles`

Para poder utilizar el módulo "fs" para trabajar con archivos, primero hay que poder nombrar el archivo con el que se quiere trabajar. Los archivos se especifican más a menudo por la *ruta*, lo que significa el nombre del propio archivo, más la jerarquía de directorios en la que aparece el archivo. Si una ruta es *absoluta*, significa que se especifican los directorios hasta la raíz del sistema de archivos. En caso contrario, la ruta es *relativa* y sólo tiene sentido en relación con alguna otra ruta, normalmente el *directorio de trabajo actual*. Trabajar con rutas puede ser un poco complicado porque los diferentes sistemas operativos utilizan diferentes caracteres para separar los nombres de los directorios, es fácil duplicar accidentalmente esos caracteres separadores cuando se concatenan las rutas, y porque los

segmentos de la ruta del directorio padre .. / necesitan un manejo especial. El módulo "path" de Node y un par de otras características importantes de Node ayudan:

```
// Algunos procesos de rutas importantes. cwd() // Ruta absoluta del directorio de trabajo actual. __filename // Ruta absoluta del archivo que contiene el código actual. __dirname // Ruta absoluta del directorio que contiene __filename. os. homedir() // El directorio personal del usuario.

const path = require("path");

path.sep // O bien "/" o bien "\ dependiendo de su sistema operativo

// El módulo path tiene funciones de análisis simples let p =
"src/pkg/test.js"; // Un ejemplo de path path.basename(p) // => "test.js"
path.extname(p) // => ".js" path.dirname(p) // => "src/pkg" path.
basename(path.dirname(p)) // => "pkg" path.dirname(path.
dirname(p)) // => "src"

// normalize() limpia las rutas: path.normalize("a/b/c/..//d/") // => "a/b/d/": maneja los segmentos .. / path.normalize("a/.//b") // => "a/b": elimina los segmentos "./" path.normalize("//a//b//") // => "/a/b/": elimina los duplicados /

// join() combina segmentos de la ruta, añadiendo separadores, y luego normaliza la ruta. join("src", "pkg", "t.js") // => "src/pkg/t.js"

// resolve() toma uno o más segmentos de ruta y devuelve un valor absoluto // ruta. Comienza con el último argumento y trabaja hacia atrás, deteniéndose // cuando ha construido una ruta absoluta o resolviendo contra process.cwd().
path.resolve() // => process.cwd() path.resolve("t.js") // =>
path.join(process.cwd(), "t.js") path.resolve("/tmp", "t.js") // =>
"/tmp/t.js"

path.resolve("/a", "/b", "t.js") // => "/b/t.js"
```

Tenga en cuenta que `path.normalize()` es simplemente una función de manipulación de cadenas que no tiene acceso al sistema de archivos real. El

Las funciones `fs.realpath()` y `fs.realpathSync()` realizan canonización con conocimiento del sistema de archivos: resuelven los enlaces simbólicos e interpretan los nombres de ruta relativos en relación con el directorio de trabajo actual.

En los ejemplos anteriores, hemos asumido que el código se ejecuta en un sistema operativo basado en Unix y que `path.sep` es `"/"`. Si quiere trabajar con rutas de estilo Unix aunque esté en un sistema Windows, entonces utilice `path.posix` en lugar de `path`. Y a la inversa, si quiere trabajar con rutas de Windows aunque esté en un sistema Unix, use `path.win32`. `path.posix` y `path.win32` definen las mismas propiedades y funciones que el propio `path`.

Algunas de las funciones "fs" que cubriremos en las próximas secciones esperan un *descriptor de archivo* en lugar de un nombre de archivo. Los descriptores de archivo son números enteros que se utilizan como referencias a nivel de sistema operativo para los archivos "abiertos". Se obtiene un descriptor para un nombre dado llamando a la función `fs.open()` (o `fs.openSync()`). Los procesos sólo pueden tener un número limitado de archivos abiertos a la vez, por lo que es importante que llame a `fs.close()` en sus descriptores de archivo cuando haya terminado con ellos. Necesitas abrir archivos si quieres usar las funciones de más bajo nivel `fs.read()` y `fs.write()` que te permiten saltar dentro de un archivo, leyendo y escribiendo trozos de él en diferentes momentos. Hay otras funciones en el módulo "fs" que utilizan descriptores de archivo, pero todas tienen versiones

basadas en nombres, y sólo tiene sentido utilizar las funciones basadas en descriptores si vas a abrir el archivo para leer o escribir de todos modos.

Por último, en la API basada en promesas definida por `fs.promises`, el equivalente de `fs.open()` es `fs.promises.open()`, que devuelve una promesa que resuelve un objeto `FileHandle`. Este objeto `FileHandle` tiene el mismo propósito que un descriptor de archivo. De nuevo, sin embargo, a menos que necesites usar los métodos de `lectura()` y `escritura()` de más bajo nivel de un `FileHandle`, no hay realmente ninguna razón para crear uno. Y si crea un `FileHandle`, debe recordar llamar a su método `close()` una vez que haya terminado con él.

### 16.7.2 Lectura de archivos

Node permite leer el contenido de los archivos de una sola vez, a través de un stream, o con la API de bajo nivel.

Si sus archivos son pequeños, o si el uso de la memoria y el rendimiento no son la mayor prioridad, entonces a menudo es más fácil leer todo el contenido de un archivo con una sola llamada. Puede hacerlo de forma sincrónica, con una devolución de llamada, o con una Promesa. Por defecto, obtendrá los bytes del archivo como un buffer, pero si especifica una codificación, obtendrá una cadena decodificada en su lugar.

```

const fs = require("fs");
let buffer = fs. readFileSync("test.data"); // Sincrónico, devuelve el buffer
let text = fs. readFileSync("data.csv", "utf8"); // Sincrónico, devuelve la cadena

// Leer los bytes del archivo de forma asíncrona fs.
readFile("test.data", (err, buffer) => { if (err) {

    // Maneja el error aquí
} si no {
    // Los bytes del archivo están en el buffer
}
});

// Lectura asíncrona basada en promesas fs.
promises . readFile("data.csv", "utf8")
    . entonces(procesarArchivoTexto)
    . catch(handleReadError);

// O utilizar la API Promise con await dentro de una función async async function
processText(filename, encoding="utf8") { let text = await fs. promises.
readFile(filename, encoding); // ... procesar el texto aquí...
}

```

Si puedes procesar el contenido de un archivo de forma secuencial y no necesitas tener todo el contenido del archivo en memoria al mismo tiempo, entonces leer un archivo a través de un stream puede ser el enfoque más eficiente. Hemos cubierto los flujos extensamente: aquí está cómo usted podría usar un flujo y el método pipe() para escribir el contenido de un archivo a la salida estándar:

```

function printFile(filename, encoding="utf8") { fs.
createReadStream(filename, encoding). pipe(process. stdout); }

```

Por último, si necesita un control de bajo nivel sobre exactamente qué bytes lee de un archivo y cuándo los lee, puede abrir un archivo para obtener un descriptor de archivo y luego utilizar `fs.read()`, `fs.readSync()` o `fs.promises.read()` para leer un número específico de bytes desde una ubicación de origen especificada del archivo en un búfer especificado en la posición de destino especificada:

```
const fs = require("fs");

// Leer una porción específica de un archivo de datos fs.
open("data", (err, fd) => { if (err) { // Informar del error de
    alguna manera return; } try {
    // Leer los bytes 20 a 420 en un buffer recién asignado. fs. read(fd, Buffer.
    alloc(400), 0, 400, 20, (err, n, b) => {
        // err es el error, si lo hay.
        // n es el número de bytes realmente leídos
        // b es el buffer en el que se leyeron los bytes.      });
    } finally { // Usa una
    cláusula finally para que siempre fs. close(fd); // cierre el descriptor de archivo abierto
    }
});
```

La API `read()` basada en callbacks es incómoda de usar si necesitas leer más de un trozo de datos de un archivo. Si puedes utilizar la API sincrónica (o la API basada en promesas con `await`), resulta fácil leer varios trozos de un archivo:

```
const fs = require("fs");

function readData(filename) { let fd = fs. openSync(filename); try {
    // Leer la cabecera del archivo
```

```

let header = Buffer.alloc(12); // Un buffer de 12 bytes fs.readSync(fd, header, 0,
12, 0);

// Verificar el número mágico del archivo let magic = header.
readInt32LE(0); if (magic !== 0xDADAFEE) { throw new Error("El archivo es
de tipo incorrecto"); }

// Ahora obtenemos el offset y la longitud de los datos de la cabecera let offset =
header.readInt32LE(4); let length = header.readInt32LE(8);

// Y leer esos bytes del archivo let data = Buffer.alloc(length); fs.
readSync(fd, data, 0, length, offset); return data;

} finalmente {
// Siempre cierra el archivo, incluso si se lanza una excepción por encima de
fs.closeSync(fd); }

}

```

### 16.7.3 Escribir archivos

Escribir archivos en Node es muy parecido a leerlos, con algunos detalles extra que debes conocer. Uno de estos detalles es que la forma de crear un nuevo archivo es simplemente escribiendo en un nombre de archivo que no exista ya.

Al igual que con la lectura, hay tres formas básicas de escribir archivos en Node. Si tienes todo el contenido del archivo en una cadena o un buffer, puedes escribirlo todo en una sola llamada con `fs.writeFile()` (basado en llamadas), `fs.writeFileSync()` (sincrónico), o `fs.promises.writeFile()` (basado en promesas):

```
fs.writeFileSync(path.resolve(__dirname, "settings.json"), JSON.stringify(settings));
```

Si los datos que está escribiendo en el archivo son una cadena, y desea utilizar una codificación distinta de "utf8", pase la codificación como un tercer argumento opcional.

Las funciones relacionadas `fs.appendFile()`, `fs.appendFileSync()`, y `fs.promises.appendFile()` son similares, pero cuando el archivo especificado ya existe, añaden sus datos al final en lugar de sobrescribir el contenido del archivo existente.

Si los datos que quieras escribir en un archivo no están todos en un trozo, o si no están todos en la memoria al mismo tiempo, entonces usar un flujo escribible es un buen enfoque, asumiendo que planeas escribir los datos desde el principio hasta el final sin saltar en el archivo:

```
const fs = require("fs"); let output = fs.createWriteStream("numbers.txt"); for(let i = 0; i < 100; i++) { output.write(`${i}\n`); } output.end();
```

Por último, si quiere escribir datos en un archivo en múltiples trozos, y quiere ser capaz de controlar la posición exacta dentro del archivo en la que se escribe cada trozo, entonces puede abrir el archivo con `fs.open()`, `fs.openSync()`, o `fs.promises.open()` y luego utilizar el descriptor de archivo resultante con las funciones `fs.write()` o `fs.writeFileSync()`. Estas funciones vienen en diferentes formas para cadenas y buffers. La variante de cadena toma un descriptor de archivo, una cadena y la posición del archivo en la que se escribirá esa cadena (con una codificación como cuarto argumento opcional). La variante de buffer toma un descriptor de archivo, un buffer, un offset y una longitud que especifican un

trozo de datos dentro del buffer, y una posición de archivo en la que escribir los bytes de ese trozo. Y si tienes un array de objetos Buffer que quieras escribir, puedes hacerlo con una sola función `fs.writev()` o `fs.writevSync()`. Existen funciones similares de bajo nivel para escribir buffers y cadenas usando `fs.promises.open()` y el objeto `FileHandle` que produce.

### CADENAS DE MODO DE

Nosvio el `fs.open()` y `fs.openSync()` antes al utilizar la API de bajo nivel para leer archivos. En ese caso de uso, bastaba con pasar el nombre del archivo a la función de apertura. Cuando se quiere para escribir un archivo, sin embargo, también debe especificar un segundo argumento de cadena que especifique cómo pi para utilizar el descriptor de archivo. Algunas de las cadenas de banderas disponibles son las

**"w"**

Abrir el archivo para escribir

**"w+"**

Abierto para escribir y leer

**"wx"**

Abrir para crear un nuevo archivo; falla si el archivo nombrado ya existe

**"wx+"**

Abrir para crear, y también permitir la lectura; falla si el archivo nombrado ya existe

**"a"**

Abrir el archivo para añadirlo; el contenido existente no se sobrescribirá

"a+"

Abierto para añadir, pero también permite la lectura

Si no pasa una de estas cadenas de bandera `fs.open()` o `fs.openSync()` utilizan el valor por defecto "r", haciendo que el descriptor de archivo sea de sólo lectura. Tenga en cuenta que también puede ser útil pasar estas banderas como argumentos a los métodos de escritura de

```
// Escribir en un archivo en una sola llamada, pero añadir a todo lo que ya está allí.  
// Esto funciona como fs.appendFileSync()  
fs.writeFileSync("mensajes.log", "holo", { ban: "a" });  
  
// Abrir un flujo de escritura, pero lanzar un error si el archivo ya existe.  
// ¡No queremos sobrescribir algo accidentalmente!  
// Tenga en cuenta que la opción anterior es "bandera" y es "banderas" aquí  
fs.createWriteStream("mensajes.log", { band: "wx" });
```

Puedes cortar el final de un archivo con `fs.truncate()`, `fs.truncateSync()`, o `fs.promises.truncate()`. Estos toman una ruta como primer argumento y una longitud como segundo, y modifican el archivo para que tenga la longitud especificada. Si se omite la longitud, se utiliza cero y el fichero queda vacío. A pesar del nombre de estas funciones, también pueden utilizarse para ampliar un archivo: si se especifica una longitud mayor que el tamaño actual del archivo, éste se amplía con cero bytes hasta el nuevo tamaño. Si ya ha abierto el archivo que desea modificar, puede utilizar `ftruncate()` o `ftruncateSync()` con el descriptor de archivo o `FileHandle`.

Las diversas funciones de escritura de archivos descritas aquí devuelven o invocan su callback o resuelven su Promesa cuando los datos han sido "escritos" en el sentido de que Node los ha entregado al sistema operativo. Pero esto no significa necesariamente que los datos se hayan escrito realmente en el almacenamiento persistente: al menos algunos de sus datos

pueden estar todavía en el búfer en algún lugar del sistema operativo o en un controlador de dispositivo a la espera de ser escritos en el disco. Si llama a `fs.writeFileSync()` para escribir sincrónicamente algunos datos en un archivo, y si hay un corte de energía inmediatamente después de que la función retorne, aún puede perder datos. Si quiere forzar la salida de sus datos al disco para saber con seguridad que se han guardado de forma segura, utilice `fs.fsync()` o `fs.fsyncSync()`. Estas funciones sólo funcionan con descriptores de archivo: no existe una versión basada en la ruta.

#### 16.7.4 Operaciones con archivos

La discusión anterior sobre las clases de flujo de Node incluyó dos ejemplos de funciones `copyFile()`. Estas no son utilidades prácticas que usted usaría realmente porque el módulo "fs" define su propio método `fs.copyFile()` (y también `fs.copyFileSync()` y `fs.promises.copyFile()`, por supuesto).

Estas funciones toman el nombre del archivo original y el nombre de la copia como sus dos primeros argumentos. Estos pueden ser especificados como cadenas o como objetos URL o Buffer. Un tercer argumento opcional es un entero cuyos bits especifican banderas que controlan los detalles de la operación de copia. Y en el caso de `fs.copyFile()`, basado en la devolución de llamada, el argumento final es una función de devolución de llamada que se llamará sin argumentos cuando se complete la copia, o que se llamará con un argumento de error si algo falla. A continuación se muestran algunos ejemplos:

```

// Copia básica de archivos sincrónica. fs. copyFileSync("ch15.txt", "ch15.bak");

// El argumento COPYFILE_EXCL copia sólo si el nuevo archivo no // existe ya. Evita que
// las copias sobrescriban los archivos existentes.
fs. copyFile("ch15.txt", "ch16.txt", fs. constants. COPYFILE_EXCL, err => {

    // Esta llamada de retorno será llamada cuando termine. En caso de error, err será no
    nulo. });

// Este código demuestra la versión basada en promesas de la función copyFile.
// Dos banderas se combinan con el operador de bitwize OR |. Las banderas significan
// que
// los archivos existentes no se sobrescribirán, y que si el sistema de archivos soporta
// la copia será un clon de copia sobre escritura del archivo original, es decir
// que no se necesitará espacio de almacenamiento adicional hasta que se
// modifique el original // o la copia. fs. promises. copyFile("Datos importantes",
// 'Datos importantes ${new Date().toISOString()}' fs.constants.COPYFILE_EXCL |
// fs.constants.COPYFILE_FICLONE)
    .then(() => {
        console.log("Copia de seguridad completa");
    });
    .catch(err => {
        console.error("Copia de seguridad fallida", err);
    });
}

```

La función fs.rename() (junto con las variantes habituales sincrónicas y basadas en promesas) mueve y/o renombra un archivo. Llámela con la ruta actual del archivo y la nueva ruta deseada para el archivo. No hay argumento de banderas, pero la versión basada en callback toma un callback como tercer argumento:

```
fs. renameSync("ch15.bak", "backups/ch15.bak");
```

Tenga en cuenta que no hay ninguna bandera para evitar que el renombramiento sobrescriba un archivo existente. También tenga en cuenta que los archivos sólo pueden ser renombrados dentro de un sistema de archivos.

Las funciones `fs.link()` y `fs.symlink()` y sus variantes tienen las mismas firmas que `fs.rename()` y se comportan de forma similar a `fs.copyFile()`, excepto que crean enlaces duros y enlaces simbólicos, respectivamente, en lugar de crear una copia.

Finalmente, `fs.unlink()`, `fs.unlinkSync()`, y `fs.promises.unlink()` son las funciones de Node para borrar un archivo. (La denominación poco intuitiva se hereda de Unix, donde borrar un archivo es básicamente lo contrario de crear un enlace duro a él). Llame a esta función con la cadena, el búfer o la ruta URL del archivo que va a ser eliminado, y pase una llamada de retorno si está utilizando la versión basada en llamadas de retorno:

```
fs.unlinkSync("backups/ch15.bak");
```

### 16.7.5 Metadatos de los archivos

Las funciones `fs.stat()`, `fs.statSync()` y `fs.promises.stat()` permiten obtener los metadatos de un archivo o directorio específico.

Por ejemplo:

```
const fs = require("fs"); let stats = fs.statSync("book/ch15.md"); stats.isFile() //=> true: es un archivo ordinario stats.isDirectory() //=> false: no es un directorio stats.size // tamaño del archivo en bytes stats.atime // tiempo de acceso: Fecha de la última lectura stats.mtime // tiempo de modificación: Fecha en la que se escribió por última vez stats.uid // el id de usuario del propietario del archivo stats.gid // el id de grupo del propietario del archivo stats.mode.toString(8) // los permisos del archivo, en forma de cadena octal
```

El objeto Stats devuelto contiene otras propiedades y métodos más oscuros, pero este código demuestra los que es más probable que utilice.

fs.lstat() y sus variantes funcionan igual que fs.stat(), excepto que si el archivo especificado es un enlace simbólico, Node devolverá los metadatos del propio enlace en lugar de seguir el enlace.

Si ha abierto un archivo para producir un descriptor de archivo o un objeto FileHandle, entonces puede utilizar fs.fstat() o sus variantes para obtener información de metadatos para el archivo abierto sin tener que especificar el nombre del archivo de nuevo.

Además de consultar los metadatos con fs.stat() y todas sus variantes, también existen funciones para modificar los metadatos.

fs.chmod(), fs.lchmod() y fs.fchmod() (junto con las versiones sincrónicas y basadas en promesas) establecen el "modo" o los permisos de un archivo o directorio. Los valores de modo son enteros en los que cada bit tiene un significado específico y es más fácil pensar en ellos en notación octal. Por ejemplo, para hacer que un archivo sea de sólo lectura para su propietario e inaccesible para todos los demás, utilice 0o400:

```
fs.chmodSync("ch15.md", 0o400); // No lo borres accidentalmente!
```

fs.chown(), fs.lchown() y fs.fchown() (junto con Sincrónico y basado en promesas) establecen el propietario y el grupo (como IDs) para un archivo o directorio. (Estos importan porque interactúan con los permisos de archivo establecidos por fs.chmod().)

Por último, puede establecer el tiempo de acceso y el tiempo de modificación de un archivo o directorio con `fs.utimes()` y `fs.futimes()` y sus variantes.

### 16.7.6 Trabajar con directorios

Para crear un nuevo directorio en Node, utilice `fs.mkdir()`, `fs.mkdirSync()`, o `fs.promises.mkdir()`. La primera es la ruta del directorio que se va a crear. El segundo argumento opcional puede ser un entero que especifica el modo (bits de permisos) para el nuevo directorio. O puede pasar un objeto con propiedades opcionales de modo y recursividad. Si la recursividad es verdadera, esta función creará cualquier directorio en la ruta que no exista ya:

```
// Asegúrese de que dist/ y dist/lib/ existen. fs. mkdirSync("dist/lib", { recursive: true });
```

`fs.mkdtemp()` y sus variantes toman un prefijo de ruta que usted proporciona, le añaden algunos caracteres aleatorios (esto es importante para la seguridad), crean un directorio con ese nombre y le devuelven (o pasan a una llamada de retorno) la ruta del directorio.

Para eliminar un directorio, utilice `fs.rmdir()` o una de sus variantes. Tenga en cuenta que los directorios deben estar vacíos antes de poder ser eliminados:

```
// Crear un directorio temporal aleatorio y obtener su ruta, entonces
// borrarlo cuando hayamos terminado
let tempDirPath; try {
  tempDirPath = fs. mkdtempSync(path. join(os. tmpdir(),
  "d"));
}

// Haz algo con el directorio aquí
```

```
} finally { // Eliminar el directorio temporal cuando hayamos terminado con él fs.  
rmdirSync(tempDirPath); }
```

El módulo "fs" proporciona dos APIs distintas para listar el contenido de un directorio. En primer lugar, `fs.readdir()`, `fs.readdirSync()` y `fs.promises.readdir()` leen todo el directorio de una vez y le dan una matriz de cadenas o una matriz de objetos `Dirent` que especifican los nombres y tipos (archivo o directorio) de cada elemento. Los nombres de archivo devueltos por estas funciones son sólo el nombre local del archivo, no la ruta completa.

He aquí algunos ejemplos:

```
let tempFiles = fs.readdirSync("/tmp"); // devuelve una matriz de cadenas  
  
// Utilice la API basada en promesas para obtener una matriz de Dirent, y luego  
// imprimir las rutas de los subdirectorios fs.promises.readdir("/tmp", {withFileTypes:  
true}) . then(entries => { entries.filter(entry => entry.isDirectory()) . map(entry =>  
entry.name) . forEach(name => console.log(path.join("/tmp/", name)); })  
 . catch(console.error);
```

Si prevé la necesidad de listar directorios que pueden tener miles de entradas, puede preferir el enfoque de flujo de `fs.opendir()` y sus variantes. Estas funciones devuelven un objeto `Dir` que representa el directorio especificado. Puede utilizar los métodos `read()` o `readSync()` del objeto `Dir` para leer un `Dirent` a la vez. Si pasas una función de devolución de llamada a `read()`, ésta llamará a la devolución de llamada.

Y si omites el argumento `callback`, devolverá una Promesa. Cuando no haya más entradas de directorio, obtendrá `null` en lugar de un objeto `Dirent`.

La forma más sencilla de utilizar los objetos Dir es como iteradores asíncronos con un bucle for/await. Aquí, por ejemplo, hay una función que utiliza la API de streaming para listar las entradas del directorio, llama a stat() en cada entrada, e imprime los nombres y tamaños de los archivos y directorios:

```
const fs = require("fs"); const path =
require("path");

async function listDirectory(dirpath) { let dir = await fs.promises.
opendir(dirpath); for await (let entry of dir) { let name = entry.name; if
(entry.isDirectory()) { name += "/"; // Añadir una barra al final de los
subdirectorios }

let stats = await fs.promises.stat(path.join(dirpath, name)); let size = stats.size;
console.log(String(size).padStart(10), name); }

}
```

## 16.8 Clientes y servidores HTTP

Los módulos "http", "https" y "http2" de Node son implementaciones completas pero de nivel relativamente bajo de los protocolos HTTP. Definen APIs completas para implementar clientes y servidores HTTP. Debido a que las APIs son relativamente de bajo nivel, no hay espacio en este capítulo para cubrir todas las características. Pero los ejemplos que siguen demuestran cómo escribir clientes y servidores básicos.

La forma más sencilla de realizar una petición HTTP GET básica es con http.get() o https.get(). El primer argumento de estas funciones es la URL que se desea obtener. (Si se trata de una URL http://, debe utilizar el módulo "http", y si es una URL https://

debe utilizar el módulo "https"). El segundo argumento es un callback que será invocado con un objeto IncomingMessage cuando la respuesta del servidor haya comenzado a llegar. Cuando se llama al callback, el estado HTTP y las cabeceras están disponibles, pero el cuerpo puede no estar listo todavía. El objeto IncomingMessage es un flujo legible, y puede utilizar las técnicas demostradas anteriormente en este capítulo para leer el cuerpo de la respuesta desde él.

La función getJSON() al final de §13.2.6 utilizaba la función http.get() como parte de una demostración del constructor Promise(). Ahora que conoce los flujos de Node y el modelo de programación de Node en general, vale la pena volver a ver ese ejemplo para ver cómo se utiliza http.get().

http.get() y https.get() son variantes ligeramente simplificadas de las funciones más generales http.request() y https.request(). La siguiente función postJSON() demuestra cómo utilizar https.request() para realizar una solicitud HTTPS POST que incluya un cuerpo de solicitud JSON. Al igual que la función getJSON() del Capítulo 13, espera una respuesta JSON y devuelve una Promise que cumple con la versión analizada de esa respuesta:

```
const https = require("https");

/*
 *      Convierte el objeto body en una cadena JSON y luego lo envía por HTTPS POST
 * al
 *      punto final de la API especificado en el host especificado. Cuando llega la
 * respuesta,
 *      parsear el cuerpo de la respuesta como JSON y resolver el
 * Promesa con * ese valor
 * analizado.
 */
function postJSON(host, endpoint, body, port, username, password) {
    // Devuelve un objeto Promise inmediatamente, luego llama a resolve o reject //
    // cuando la petición HTTPS tiene éxito o falla. return new Promise((resolve, reject) => {
    // Convierte el objeto body en una cadena let bodyText = JSON.stringify(body);

    // Configurar la petición HTTPS let requestOptions = { method: "POST", // O
    // "GET", "PUT", "DELETE", etc. host: host, // El host al que conectarse path: endpoint,
    // // La ruta de la URL headers: { // Las cabeceras HTTP para la solicitud "Content-
    // Type": "application/json",
    // "Content-Length": Buffer.byteLength(bodyText)
    }    };

    if (port) { // Si se especifica un puerto, requestOptions. port = port; // lo utiliza
    para la solicitud.    }
    // Si se especifican las credenciales, añada una cabecera de Autorización.
    if (username && password) { requestOptions. auth = `${username}:${password}`;
    }
}
```

```
// Ahora crea la solicitud basada en el objeto de configuración let request = https.  
request(requestOptions);  
  
// Escribe el cuerpo de la petición POST y finaliza la petición.  
request.write(bodyText); request.end();  
  
// Falla en los errores de la solicitud (como la falta de conexión a la red)  
request.on("error", e => reject(e));  
  
// Manejar la respuesta cuando empieza a llegar.  
request.on("response", response => { if (response.  
statusCode !== 200) { reject(new Error(`HTTP status  
${response.statusCode}`));  
// No nos importa el cuerpo de la respuesta en este caso, pero  
// no queremos que se quede en un buffer en algún lugar, así que  
// ponemos el flujo en modo fluido sin registrar // un manejador de  
"datos" para que el cuerpo sea descartado.  
response.resume(); return; }  
  
// Queremos texto, no bytes. Asumimos que el texto será  
// con formato JSON pero no se molestan en comprobar la cabecera //  
Content-Type. response.setEncoding("utf8");  
  
// Node no tiene un analizador JSON de flujo, así que leemos el // cuerpo  
completo de la respuesta en una cadena. let body = ""; response.on("data", chunk => {  
body += chunk; });  
  
// Y ahora maneja la respuesta cuando está completa.
```

```
    response.on("end", () => { // Cuando la respuesta está hecha, try { // intenta
      parsearla como JSON resolve(JSON.parse(body)); // y resuelve el resultado.      }
      catch(e) { // O, si algo va mal, reject(e); // rechazar con el error }
    });
  });
}
}
```

Además de realizar peticiones HTTP y HTTPS, los módulos "http" y "https" también permiten escribir servidores que respondan a esas peticiones. El enfoque básico es el siguiente:

- Crea
  - un nuevo objeto Servidor.
  - Llama a su método listen() para comenzar a escuchar peticiones en un puerto especificado.
- Registra un manejador de eventos para los eventos "request", usa ese manejador para leer la solicitud del cliente (particularmente la propiedad request.url), y escribe tu respuesta.

El código que sigue crea un simple servidor HTTP que sirve archivos estáticos desde el sistema de archivos local y también implementa un punto final de depuración que responde a la solicitud de un cliente haciéndose eco de dicha solicitud.

```
// Este es un simple servidor HTTP estático que sirve archivos de un
//. También implementa un /test/mirror especial
```

*endpoint que // se hace eco de la solicitud entrante, lo que puede ser útil cuando se depuran los clientes.*

```
const http = require("http"); // Utiliza "https" si tienes un certificado
const url = require("url"); // Para analizar URLs
const path = require("path"); // Para manipular las rutas del sistema de archivos
const fs = require("fs"); // Para leer archivos
```

*// Sirve los archivos del directorio raíz especificado a través de un servidor HTTP que // escucha en el puerto especificado.*

```
function serve(rootDirectory, port) {
  let server = new http.Server();
  server.listen(port);
  console.log("Listening on port", port);
```

*// Cuando lleguen las peticiones, gestíonalas con esta función server.on("request", (request, response) => {*

```
  // Obtener la parte de la ruta de la URL de la solicitud, ignorando
  // cualquier parámetro de consulta que se le añada. let endpoint = url.parse(request.url).pathname;
```

*// Si la petición era para "/test/mirror", devuelve la petición // literalmente. Útil cuando se necesita ver las cabeceras y el cuerpo de la petición.*

```
  if (endpoint === "/test/mirror") {
    response.setHeader("Content-Type", "text/plain; charset=UTF-8");
```

*// Especifica el código de estado de la respuesta response.writeHead(200); // 200 OK*

```
  write(`${request.method} ${request.url} HTTP/${request.httpVersion}`);
}
```

```
// La salida de las cabeceras de la solicitud let headers = request.  
rawHeaders; for(let i = 0; i < headers.length; i += 2) { response.  
write(`${headers[i]}: ${headers[i+1]}\r\n`);  
}  
  
// Termina las cabeceras con una línea extra en blanco response.write("\r\n");  
  
// Ahora necesitamos copiar cualquier cuerpo de la solicitud al cuerpo de la  
respuesta // Como ambos son flujos, podemos usar una solicitud de tubería.  
tubería(response);  
} // En caso contrario, sirve un archivo del directorio local. else {  
// Asignar el punto final a un archivo en el sistema de archivos local let  
filename = endpoint.substring(1); // quitar la parte inicial /  
// No permita "../" en la ruta porque sería un agujero de seguridad // para servir  
cualquier cosa fuera del directorio raíz.  
filename = filename.replace(/\.\.\//g, ""); // Ahora convierta de relativo a  
absoluto filename = path.resolve(rootDirectory, filename);  
  
// Ahora adivina el tipo de contenido del archivo basado en la extensión let type;  
switch(path.extname(filename)) { case ".html":  
    case ".htm": type = "text/html"; break; case ".js": type = "text/javascript";  
    break; case ".css": type = "text/css"; break; case ".png": type = "image/png"; break;  
    case ".txt": type = "text/plain"; break; por defecto: type = "application/octet-  
stream"; break;  
}
```

```

let stream = fs.createReadStream(filename); stream.once("readable", () => {
  // Si el flujo se vuelve legible, entonces establece el
  // Cabecera Content-Type y un estado 200 OK.
  A continuación, canalice el
    // flujo del lector de archivos a la respuesta. La tubería // llamará
    // automáticamente a response.end() cuando el flujo termine.
    response.setHeader("Content-Type", type); response.writeHead(200);
  stream.pipe(response); });

  stream.on("error", (err) => {
    // En su lugar, si obtenemos un error al intentar abrir el flujo // entonces el
    // archivo probablemente no existe o no es legible.
    // Enviar una respuesta de texto plano 404 Not Found con el // mensaje de
    // error.
    response.setHeader("Content-Type", "text/plain;
  charset=UTF-8"); response.writeHead(404); response.end(err.message);
  });
}

// Cuando se nos invoca desde la línea de comandos, llama a la función serve()
serve(process.argv[2] || "/tmp", parseInt(process.argv[3]) ||
8000);

```

Los módulos incorporados de Node son todo lo que necesitas para escribir servidores HTTP y HTTPS simples. Sin embargo, ten en cuenta que los servidores de producción no suelen construirse directamente sobre estos módulos. En su lugar, la mayoría de los servidores no triviales se implementan utilizando bibliotecas externas -como el framework Express- que proporcionan

"middleware" y otras utilidades de alto nivel que los desarrolladores web de backend han llegado a esperar.

## 16.9 Servidores y clientes de red no HTTP

Los servidores y clientes web se han vuelto tan omnipresentes que es fácil olvidar que es posible escribir clientes y servidores que no utilicen

HTTP. Aunque Node tiene fama de ser un buen entorno para escribir servidores web, también tiene soporte completo para escribir otros tipos de servidores y clientes de red.

Si te sientes cómodo trabajando con streams, entonces la creación de redes es relativamente sencilla, porque los sockets de red son simplemente una clase de stream dúplex. El módulo "net" define las clases Server y Socket. Para crear un servidor, llame a `net.createServer()`, y luego llame a `listen()` del objeto resultante para decirle al servidor en qué puerto debe escuchar las conexiones. El objeto Servidor generará eventos de "conexión" cuando un cliente se conecte en ese puerto, y el valor que se pase al escuchador de eventos será un objeto Socket. El objeto Socket es un flujo dúplex, y puedes usarlo para leer datos del cliente y escribir datos al cliente. Llame a `end()` en el Socket para desconectarse.

Escribir un cliente es aún más fácil: pasa un número de puerto y un nombre de host a `net.createConnection()` para crear un socket que se comunique con cualquier servidor que se esté ejecutando en ese host y que escuche en ese puerto. A continuación, utilice ese socket para leer y escribir datos desde y hacia el servidor.

El siguiente código demuestra cómo escribir un servidor con el módulo "net". Cuando el cliente se conecta, el servidor cuenta una broma de toc-toc:

```
// Un servidor TCP que entrega bromas interactivas en el puerto 6789. (¿Por qué el seis teme al siete? ¡Porque el siete se comió al nueve!) const net = require("net");
const readline = require("readline");

// Crear un objeto Servidor y comenzar a escuchar conexiones let server = net.createServer();
server.listen(6789, () => console.log("Delivering laughs on port 6789"));

// Cuando un cliente se conecte, dile un chiste de toc-toc. servidor.
on("conexión", socket => { tellJoke(socket)
  .then(() => socket.end()) // Cuando la broma haya terminado, cierra el socket.
  .catch((err) => { console.error(err); // Registra cualquier error que ocurra, socket.end(); // ipero cierra el socket! });
});

// Estos son todos los chistes que conocemos.
const jokes = { "Boo": "No llores... ¡es sólo una broma!",
  "Lechuga": "¡Déjanos entrar! Hace mucho frío aquí fuera!",
  "Una ancianita": "¡Vaya, no sabía que pudieras cantar a gritos!" };

// Realiza de forma interactiva una broma de golpe sobre este socket, sin bloquear.
async function tellJoke(socket) { // Elige uno de los chistes al azar let randomElement = a => a[Math.floor(Math.random() * a.length)]; let who = randomElement(Object.keys(jokes)); let punchline = jokes[who];
```

```
// Utilice el módulo readline para leer la entrada del usuario una línea a la vez.
let lineReader = readline.createInterface({ input: socket, output:
socket, prompt: ">> " });

// Una función de utilidad para dar salida a una línea de texto al cliente // y luego
// (por defecto) mostrar un prompt.
function output(text, prompt=true) { socket.
write(`#${text}\r\n`); if (prompt) lineReader.prompt(); }

// Las bromas Knock-knock tienen una estructura de llamada y respuesta. //
// Esperamos diferentes entradas del usuario en diferentes etapas y // tomamos
// diferentes acciones cuando obtenemos esa entrada en diferentes etapas. let stage = 0;

// Comienza el chiste de toc-toc de la manera tradicional. output("¡Toc-toc!");

// Ahora lee las líneas de forma asíncrona desde el cliente hasta que la broma
// termine.
for await (let inputLine of lineReader) { if (stage === 0) { if (inputLine.toLowerCase()
== "¿quién está ahí?") {
    // Si el usuario da la respuesta correcta en la etapa 0 // entonces cuenta la
    // primera parte del chiste y pasa a la etapa 1.
    output(who); stage = 1;
} } si no {
    // De lo contrario, enseñe al usuario a hacer bromas de toc-toc. output('Por
    favor, escriba "¿Quién está ahí?".');
}
}
```

```

} else if (stage === 1) { if (inputLine.toLowerCase() ===
`"${who.toLowerCase()} who?`) {
    // Si la respuesta del usuario es correcta en la etapa
    1, entonces
        // entregar el remate y volver ya que el chiste está hecho.
        output(` ${punchline}`, false); return;
    } else { // Hacer que el usuario siga el juego.
        output(`Por favor, escriba "${who} who?"`); }
}
}
}

```

Los servidores simples basados en texto como éste no suelen necesitar un cliente personalizado. Si la utilidad nc ("netcat") está instalada en su sistema, puede utilizarla para comunicarse con este servidor de la siguiente manera:

```

$ nc localhost 6789 ¡Toc toc!
¿Quién está ahí?
Una ancianita >> ¿Una ancianita
qué?
¡Vaya, no sabía que podías cantar a gritos!

```

Por otro lado, escribir un cliente personalizado para el servidor de bromas es fácil en Node. Simplemente nos conectamos al servidor, luego canalizamos la salida del servidor a stdout y canalizamos stdin a la entrada del servidor:

```

// Conéctate al puerto de broma (6789) en el servidor nombrado en la línea de
comandos let socket = require("net").createConnection(6789, process.argv[2]);
socket.pipe(process.stdout); // Canaliza los datos del socket a stdout process.stdin.
pipe(socket); // Canaliza los datos de stdin al socket socket.on("close", () => process.
exit()); // Salir cuando el socket se cierra.

```

Además de soportar servidores basados en TCP, el módulo "net" de Node también soporta la comunicación entre procesos a través de "sockets de dominio Unix" que se identifican por una

ruta del sistema de archivos en lugar de por un número de puerto. No vamos a cubrir ese tipo de socket en este capítulo, pero la documentación de Node tiene detalles. Otras características de Node que no tenemos espacio para cubrir aquí incluyen el módulo "dgram" para clientes y servidores basados en UDP y el módulo "tls" que es a "net" como "https" es a "http". Las clases "tls.Server" y "tls.TLSSocket" permiten la creación de servidores TCP (como el servidor de bromas "knock-knock") que utilizan conexiones encriptadas SSL como lo hacen los servidores HTTPS.

## 16.10 Trabajar con procesos infantiles

Además de escribir servidores altamente concurrentes, Node también funciona bien para escribir scripts que ejecutan otros programas. En Node el módulo "child\_process" define una serie de funciones para ejecutar otros programas como procesos hijos. Esta sección muestra algunas de esas funciones, empezando por las más simples y pasando por las más complicadas.

### 16.10.1 execSync() y execFileSync()

La forma más sencilla de ejecutar otro programa es con `child_process.execSync()`. Esta función toma el comando para ejecutar como su primer argumento. Crea un proceso hijo, ejecuta un intérprete de comandos en ese proceso y utiliza el intérprete de comandos para ejecutar el comando que le pasaste. Luego se bloquea hasta que el comando (y la shell) salgan. Si el comando sale con un error, `execSync()` lanza una excepción. De lo contrario, `execSync()` devuelve cualquier salida que el comando

escriba en su flujo stdout. Por defecto este valor de retorno es un buffer, pero puede especificar una codificación en un segundo argumento opcional para obtener una cadena en su lugar. Si el comando escribe cualquier salida a stderr, esa salida simplemente se pasa al flujo stderr del proceso padre.

Así, por ejemplo, si está escribiendo un script y el rendimiento no es una preocupación, podría utilizar child\_process.execSync() para listar un directorio con un comando de shell Unix familiar en lugar de utilizar la función fs.readdirSync():

```
const child_process = require("child_process"); let listing =  
child_process.execSync("ls -l web/*.html", {encoding: "utf8"});
```

El hecho de que execSync() invoque un shell Unix completo significa que la cadena que se le pasa puede incluir múltiples órdenes separadas por punto y coma, y puede aprovechar características del shell como comodines de nombre de archivo, tuberías y redirección de salida. Esto también significa que debe tener cuidado de no pasar nunca un comando a execSync() si alguna parte de ese comando es una entrada del usuario o proviene de una fuente similar no fiable. La compleja sintaxis de los comandos del shell puede ser fácilmente subvertida para permitir a un atacante ejecutar código arbitrario.

Si no necesita las características de un shell, puede evitar la sobrecarga de iniciar un shell utilizando child\_process.execFileSync().

Esta función ejecuta un programa directamente, sin invocar un shell.

Pero como no hay un shell involucrado, no puede analizar una línea de comandos, y debes pasar el ejecutable como primer argumento y una matriz de argumentos de línea de comandos como segundo argumento:

```
let listing = child_process.execFileSync("ls", ["-l",  
"web/"],
```

### OPCIONES DEL PROCESO INFANTIL

execSync() y muchas de las otras funciones de child\_process tienen un segundo o tercer argumento opcional que especifica detalles adicionales sobre cómo debe ejecutarse el proceso hijo. La propiedad encoding de este objeto se utilizó antes para especificar que queremos que la salida del comando se entregue como una cadena en lugar de como un buffer. Otras propiedades importantes que puede especificar son las siguientes (tenga en cuenta que no todas las opciones están disponibles para todas las funciones de proceso hijo):

- cwd especifica el directorio de trabajo para el proceso hijo. Si se omite, el proceso hijo hereda el valor de process.cwd().
- env especifica las variables de entorno a las que el proceso hijo tendrá acceso. Por defecto, los procesos hijos simplemente heredan process.env, pero puedes especificar un objeto diferente si lo deseas.
- input especifica una cadena o un búfer de datos de entrada que debe utilizarse como entrada estándar del proceso hijo. Esta opción sólo está disponible para las funciones síncronas que no devuelven un objeto ChildProcess.
- maxBuffer especifica el número máximo de bytes de salida que serán recogidos por las funciones exec. (No se aplica a spawn() y fork(), que utilizan flujos). Si un proceso hijo produce más salida que esto, será matado y saldrá con un error.
- shell especifica la ruta a un ejecutable de shell o true. Para las funciones del proceso hijo que normalmente ejecutan un comando shell, esta opción permite especificar qué shell utilizar. Para las funciones que normalmente no utilizan un shell, esta opción permite especificar que se debe utilizar un shell (estableciendo la propiedad a true) o especificar exactamente qué shell utilizar.
- El tiempo de espera especifica el número máximo de milisegundos que el proceso hijo puede ejecutar. Si no ha salido antes de que transcurra este tiempo, será matado y saldrá con un error. (Esta opción se aplica a las funciones exec pero no a spawn() o fork().)
- uid especifica el ID de usuario (un número) bajo el cual se debe ejecutar el programa. Si el proceso padre se ejecuta con una cuenta privilegiada, puede utilizar esta opción para ejecutar el proceso hijo con privilegios reducidos.

```
{codificación: "utf8"});
```

## 16.10.2 exec() y execFile()

Las funciones `execSync()` y `execFileSync()` son, como sus nombres indican, síncronas: se bloquean y no regresan hasta que el proceso hijo sale. Utilizar estas funciones es muy parecido a escribir comandos Unix en una ventana de terminal: permiten ejecutar una secuencia de comandos de uno en uno. Pero si estás escribiendo un programa que necesita realizar una serie de tareas, y esas tareas no dependen unas de otras de ninguna manera, entonces puedes querer paralelizarlas y ejecutar múltiples comandos al

mismo tiempo. Puede hacerlo con las funciones asíncronas `child_process.exec()` y `child_process.execFile()`.

`exec()` y `execFile()` son como sus variantes síncronas, excepto que devuelven inmediatamente un objeto `ChildProcess` que representa el proceso hijo en ejecución, y toman una llamada de retorno de error como argumento final. La llamada de retorno se invoca cuando el proceso hijo sale, y en realidad se llama con tres argumentos. El primero es el error, si lo hay; será nulo si el proceso terminó normalmente. El segundo argumento es la salida recolectada que fue enviada al flujo de salida estándar del proceso hijo. Y el tercer argumento es cualquier salida que haya sido enviada al flujo de error estándar del hijo.

El objeto `ChildProcess` devuelto por `exec()` y `execFile()` permite terminar el proceso hijo, y escribirle datos (que luego puede leer de su entrada estándar). Cubriremos `ChildProcess` con más detalle cuando hablemos de la función `child_process.spawn()`.

Si planea ejecutar varios procesos hijo al mismo tiempo, entonces puede ser más fácil utilizar la versión "prometida" de `exec()` que devuelve un objeto `Promise` que, si el proceso hijo sale sin error, resuelve a un objeto con propiedades `stdout` y `stderr`. Aquí, por ejemplo, hay una función que toma una matriz de comandos de la shell como entrada y devuelve una promesa que resuelve el resultado de todos esos comandos:

```
const proceso_niño = require("proceso_niño"); const util =
require("util"); const execP = util.promisify(proceso_niño.exec);

función parallelExec(comandos) {
  // Utiliza el array de comandos para crear un array de
  Promesas let promesas = comandos.map(comando => execP(comando,
  {codificación: "utf8"}));
  // Devuelve una Promesa que cumplirá a un array del cumplimiento
  // valores de cada una de las promesas individuales. (En lugar de devolver objetos
  // con las propiedades stdout y stderr sólo devolvemos el valor stdout.) return
  Promise.all(promesas).then(outputs => outputs.map(out => out.stdout)); }
module.exports = parallelExec;
```

### 16.10.3 spawn()

Las diversas funciones exec descritas hasta ahora -tanto las síncronas como las asíncronas- están diseñadas para ser utilizadas con procesos hijos que se ejecutan rápidamente y no producen mucha salida. Incluso las funciones asíncronas exec() y execFile() no son de flujo: devuelven la salida del proceso en un solo lote, sólo después de que el proceso haya salido.

La función child\_process.spawn() le permite acceder a la salida del proceso hijo, mientras el proceso sigue en ejecución. También permite escribir datos en el proceso hijo (que verá esos datos como entrada en su flujo de entrada estándar): esto significa que es posible interactuar dinámicamente con un proceso hijo, enviándole entradas basadas en la salida que genera.

spawn() no utiliza un shell por defecto, por lo que debe invocarlo como execFile() con el ejecutable a ejecutar y una matriz

separada de argumentos de línea de comandos para pasarlle. `spawn()` devuelve un objeto `ChildProcess` como lo hace `execFile()`, pero no toma un argumento de devolución de llamada. En lugar de utilizar una función de devolución de llamada, se escucha a los eventos en el objeto `ChildProcess` y en sus flujos.

El objeto `ChildProcess` devuelto por `spawn()` es un emisor de eventos. Se puede escuchar el evento "exit" para ser notificado cuando el proceso hijo salga. Un objeto `ChildProcess` también tiene tres propiedades de flujo. `stdout` y `stderr` son flujos legibles: cuando el proceso hijo escribe en sus flujos `stdout` y `stderr`, esa salida se vuelve legible a través de los flujos de `ChildProcess`. Nótese la inversión de los nombres aquí. En el proceso hijo, "`stdout`" es un flujo de salida escribible, pero en el proceso padre, la propiedad `stdout` de un objeto `ChildProcess` es un flujo de entrada legible.

Del mismo modo, la propiedad `stdin` del objeto `ChildProcess` es un flujo `Writable`: cualquier cosa que escriba en este flujo estará disponible para el proceso hijo en su entrada estándar.

El objeto `ChildProcess` también define una propiedad `pid` que especifica el id del proceso hijo. Y define un método `kill()` que puede utilizar para terminar un proceso hijo.

#### **16.10.4 `fork()`**

`child_process.fork()` es una función especializada para ejecutar un módulo de código JavaScript en un proceso Node hijo. `fork()` espera los mismos argumentos que `spawn()`, pero el primer

argumento debe especificar la ruta a un archivo de código JavaScript en lugar de un archivo binario ejecutable.



Un proceso hijo creado con fork() puede comunicarse con el proceso padre a través de sus flujos de entrada y salida estándar, como se describe en la sección anterior para spawn(). Pero además, fork() permite otro canal de comunicación, mucho más sencillo, entre los procesos padre e hijo.

Cuando se crea un proceso hijo con fork(), se puede utilizar el método send() del objeto ChildProcess devuelto para enviar una copia de un objeto al proceso hijo. Y puedes escuchar el evento "message" en el ChildProcess para recibir mensajes del hijo. El código que se ejecuta en el proceso hijo puede utilizar process.send() para enviar un mensaje al padre y puede escuchar los eventos "message" en process para recibir mensajes del padre.

Aquí, por ejemplo, hay un código que utiliza fork() para crear un proceso hijo, luego envía a ese hijo un mensaje y espera una respuesta:

```
const child_process = require("child_process");
```

```
// Iniciar un nuevo proceso node ejecutando el código en child.js en
```

```

nuestro directorio let child = child_process.fork(`${__dirname}/child.js`);

// Enviar un mensaje al niño niño. send({x: 4, y: 3});

// Imprime la respuesta del niño cuando llega. niño. on("mensaje", mensaje => {
  console.log(mensaje.hipotenusa); // Esto debería imprimir "5"
    // Como sólo enviamos un mensaje, sólo esperamos una respuesta.
    // Después de recibirla llamamos a disconnect() para terminar la conexión // entre
    // padre e hijo. Esto permite que ambos procesos salgan limpiamente.
    niño.disconnect();
});

```

Y aquí está el código que se ejecuta en el proceso hijo:

```

// Esperar los mensajes de nuestro proceso padre. on("mensaje", mensaje => {
  // Cuando recibimos uno, hacemos un cálculo y enviamos el resultado // de vuelta
  // al padre. process.send({hypotenuse: Math.hypot(mensaje.x, mensaje.y)}); });

```

Iniciar procesos hijo es una operación costosa, y el proceso hijo tendría que estar haciendo órdenes de magnitud más computacionales antes de que tuviera sentido usar fork() y la comunicación entre procesos de esta manera. Si está escribiendo un programa que necesita ser muy sensible a los eventos entrantes y también necesita realizar cálculos que consumen tiempo, entonces podría considerar el uso de un proceso hijo separado para realizar los cálculos para que no bloqueen el bucle de eventos y reduzcan la capacidad de respuesta del proceso padre. (Aunque un hilo-véase [§16.11-puede ser](#) una mejor opción que un proceso hijo en este escenario).

El primer argumento de send() se serializará con

`JSON.stringify()` y se deserializa en el proceso hijo con `JSON.parse()`, por lo que sólo debe incluir valores que estén soportados por el formato JSON. `send()` tiene un segundo argumento especial, sin embargo, que le permite transferir objetos `Socket` y `Server` (del módulo "net") a un proceso hijo. Los servidores de red tienden a estar ligados a la E/S más que a la computación, pero si ha escrito un servidor que necesita hacer más computación de la que una sola CPU puede manejar, y si está ejecutando ese servidor en una máquina con múltiples CPUs, entonces podría usar `fork()` para crear múltiples procesos hijos para manejar las peticiones. En el proceso padre, podría escuchar los eventos de "conexión" en su objeto `Servidor`, luego obtener el objeto `Socket` de ese evento de "conexión" y enviarlo() -usando el segundo argumento especial- a uno de los procesos hijos para ser manejado. (Tenga en cuenta que esta es una solución poco probable para un escenario poco común. En lugar de escribir un servidor que bifurca los procesos hijos, probablemente sea más sencillo mantener su servidor de un solo hilo y desplegar múltiples instancias del mismo en producción para manejar la carga).

## 16.11 Hilos de trabajo

Como se explicó al principio de este capítulo, el modelo de concurrencia de Node es de un solo hilo y basado en eventos. Pero en la versión 10 y posteriores, Node permite una verdadera programación multihilo, con una API que se asemeja mucho a la API Web Workers definida por los navegadores web ([§15.13](#)). La programación multihilo tiene una merecida reputación de ser difícil. Esto se debe casi exclusivamente a la necesidad de

sincronizar cuidadosamente el acceso de los hilos a la memoria compartida. Pero los hilos de JavaScript (tanto en Node como en los navegadores) no comparten memoria por defecto, por lo que los peligros y dificultades de usar hilos no se aplican a estos "trabajadores" en JavaScript.

En lugar de utilizar memoria compartida, los hilos de trabajo de JavaScript se comunican mediante el paso de mensajes. El hilo principal puede enviar un mensaje a un hilo trabajador llamando al método `postMessage()` del objeto `Worker` que representa ese hilo. El hilo worker puede recibir mensajes de su padre escuchando los eventos "message". Y los trabajadores pueden enviar mensajes al hilo principal con su propia versión de `postMessage()`, que el padre puede recibir con su propio manejador de eventos "message". El código de ejemplo dejará claro cómo funciona esto.

Hay tres razones por las que puedes querer usar hilos de trabajo en una aplicación Node:

- Si tu aplicación necesita hacer más cálculos de los que un núcleo de la CPU puede manejar, entonces los hilos te permiten distribuir el trabajo entre los múltiples núcleos, que se han convertido en algo común en los ordenadores de hoy en día. Si estás haciendo computación científica o aprendizaje automático o procesamiento de gráficos en Node, entonces es posible que quieras usar hilos simplemente para lanzar más potencia de cálculo a tu problema.
- Incluso si su aplicación no está utilizando toda la potencia de una CPU, es posible que quiera utilizar hilos para mantener la capacidad de respuesta del hilo principal. Considere un servidor que maneja solicitudes grandes pero relativamente infrecuentes.

Supongamos que sólo recibe una petición por segundo, pero que necesita emplear medio segundo de cálculo (bloqueado por la CPU) para procesar cada petición. En promedio, estará inactivo el 50% del tiempo. Pero cuando llegan dos peticiones con pocos milisegundos de diferencia, el servidor ni siquiera podrá empezar a responder a la segunda petición hasta que el cálculo de la primera respuesta haya terminado. En cambio, si el servidor utiliza un hilo de trabajo para realizar el cálculo, el servidor puede comenzar la respuesta a ambas peticiones inmediatamente y proporcionar una mejor experiencia a los clientes del servidor. Suponiendo que el servidor tenga más de un núcleo de CPU, también puede calcular el cuerpo de ambas respuestas en paralelo, pero incluso si sólo hay un único núcleo, el uso de trabajadores sigue mejorando la capacidad de respuesta.

- En general, los workers nos permiten convertir operaciones síncronas bloqueantes en operaciones asíncronas no bloqueantes. Si estás escribiendo un programa que depende de código heredado que es inevitablemente síncrono, puedes usar workers para evitar el bloqueo cuando necesites llamar a ese código heredado.

Los hilos de trabajo no son tan pesados como los procesos hijo, pero no son ligeros. Por lo general, no tiene sentido crear un trabajador a menos que tenga un trabajo significativo que hacer. Y, en general, si tu programa no está limitado por la CPU y no tiene problemas de respuesta, entonces probablemente no necesites hilos de trabajo.

### 16.11.1 Crear trabajadores y pasar mensajes

El módulo de Node que define los trabajadores se conoce como "worker\_threads". En esta sección nos referiremos a él con el identificador threads:

```
const threads = require("worker_threads");
```

Este módulo define una clase Worker para representar un hilo trabajador, y puedes crear un nuevo hilo con el constructor `threads.Worker()`. El siguiente código demuestra el uso de este constructor para crear un trabajador, y muestra cómo pasar mensajes del hilo principal al trabajador y del trabajador al hilo principal. También demuestra un truco que permite poner el código del hilo principal y el del hilo trabajador en el mismo archivo.<sup>2</sup>

```
const threads = require("worker_threads");

// El módulo worker_threads exporta la propiedad booleana isMainThread.
// Esta propiedad es verdadera cuando Node está ejecutando el hilo principal y es
// false cuando Node está ejecutando un worker. Podemos utilizar este hecho para
// implementar // los hilos principal y trabajador en el mismo archivo. if (threads.
isMainThread) {
    // Si estamos ejecutando en el hilo principal, entonces todo lo que hacemos es
    // exportar
    // una función. En lugar de realizar un cálculo intensivo
    // tarea en el hilo principal, esta función pasa la tarea a un trabajador // y devuelve
    // una Promesa que se resolverá cuando el trabajador haya terminado.
    module.exports = function reticulateSplines(splines) { return new
Promise((resolve,reject) => {
    // Crear un trabajador que cargue y ejecute este mismo archivo de código.
    // Observa el uso de la variable especial __filename. let reticulator = new threads.
Worker(__nombredelarchivo);

    // Pasar una copia del array de splines al reticulador del trabajador.
    postMessage(splines);

    // Y luego resolver o rechazar la Promesa cuando recibimos // un mensaje o error
    // del trabajador. reticulator.on("message", resolve);
})};
}
```

```

        reticulator.on("error", reject); });
    };
} si no {
    // Si llegamos aquí, significa que estamos en el trabajador, así que registramos un
    // manejador para recibir mensajes del hilo principal. Este trabajador está diseñado
    // para recibir un solo mensaje, por lo que registramos el manejador de eventos
    // con once() en lugar de on(). Esto permite que el trabajador salga de forma natural
    // cuando su trabajo esté completo. threads.parentPort.once("message", splines => {
        // Cuando obtenemos las splines del hilo padre, hacemos un bucle // a través
        // de ellas y las reticulamos todas. for(let spline of splines) {
            // Por ejemplo, supongamos que los objetos spline suelen // tener un método
            reticulate() que hace muchos cálculos. spline.reticulate ? spline.reticulate() :
            spline.reticulado = true;
        }

        // Cuando todas las splines han sido (i por fin!) reticuladas // pasa una
        // copia al hilo principal. threads.parentPort.postMessage(splines);
    });
}

```

El primer argumento del constructor Worker() es la ruta de un archivo de código JavaScript que se va a ejecutar en el hilo. En el código anterior, hemos utilizado el identificador predefinido \_\_filename para crear un trabajador que carga y ejecuta el mismo archivo que el hilo principal. En general, sin embargo, usted pasará una ruta de archivo. Tenga en cuenta que si especifica una ruta relativa, es relativa a process.cwd(), no relativa al módulo actualmente en ejecución. Si quieras una ruta relativa al módulo

actual, utiliza algo como path.resolve(\_\_dirname, 'workers/reticulator.js').

El constructor de Worker() también puede aceptar un objeto como segundo argumento, y las propiedades de este objeto proporcionan una configuración opcional para el trabajador. Cubriremos varias de estas opciones más adelante, pero por ahora ten en cuenta que si pasas {eval: true} como segundo argumento, entonces el primer argumento de Worker() se interpreta como una cadena de código JavaScript a evaluar en lugar de un nombre de archivo:

```
nuevo threads. Worker(` const threads = require("worker_threads");
threads.parentPort.postMessage(threads.isMainThread); `, {eval: true});
on("message", console.log); // Esto imprimirá "false".
```

Node hace una copia del objeto pasado a postMessage() en lugar de compartirlo directamente con el hilo trabajador. Esto evita que el hilo del trabajador y el hilo principal comparten memoria. Se podría esperar que esta copia se hiciera con JSON.stringify() y JSON.parse() ([§11.6](#)). Pero de hecho, Node toma prestada una técnica más robusta conocida como el algoritmo de clonación estructurada de los navegadores web.

El algoritmo de clonación estructurada permite la serialización de la mayoría de los tipos de JavaScript, incluidos los objetos Map, Set, Date y RegExp y las matrices tipificadas, pero no puede, en general, copiar los tipos definidos por el entorno anfitrión de Node, como los sockets y los streams. Tenga en cuenta, sin embargo, que Buffer están parcialmente soportados: si pasas un Buffer a postMessage() será recibido como un Uint8Array, y puede ser convertido de

nuevo en un Buffer con Buffer.from(). Lea más sobre el algoritmo de clonación estructurado en "["El algoritmo de clonación estructurado"](#)".

### 16.11.2 El entorno de ejecución del trabajador

En su mayor parte, el código JavaScript en un hilo de trabajo de Node se ejecuta igual que en el hilo principal de Node. Hay algunas diferencias que debes conocer, y algunas de ellas tienen que ver con las propiedades del segundo argumento opcional del constructor Worker():

- Como hemos visto, threads.isMainThread es verdadero en el hilo principal pero siempre es falso en cualquier hilo trabajador.
- En un hilo de trabajo, puede utilizar threads.parentPort.postMessage() para enviar un mensaje al hilo principal y threads.parentPort.on para registrar los manejadores de eventos para los mensajes del hilo padre. En el hilo principal, threads.parentPort es siempre nulo.
- En un hilo trabajador, threads.workerData se establece como una copia de la propiedad workerData del segundo argumento del constructor Worker(). En el hilo principal, esta propiedad es siempre nula. Puedes utilizar esta propiedad workerData para pasar un mensaje inicial al trabajador que estará disponible tan pronto como se inicie, de modo que el trabajador no tenga que esperar a un evento "mensaje" antes de empezar a trabajar.
- Por defecto, process.env en un hilo trabajador es una copia de process.env en el hilo padre. Pero el hilo padre puede especificar un conjunto personalizado de variables de entorno estableciendo

la propiedad env del segundo argumento del constructor Worker().

Como caso especial (y potencialmente peligroso), el subprocesso

padre puede establecer la propiedad env a

threads.SHARE\_ENV, que hará que los dos hilos comparten un único conjunto de variables de entorno para que un cambio en un hilo sea visible en el otro.

- Por defecto, el flujo process.stdin en un trabajador nunca tiene datos legibles en él. Puedes cambiar este valor por defecto pasando stdin: true en el segundo argumento del constructor Worker(). Si lo haces, entonces la propiedad stdin del objeto Worker es un flujo Writable. Cualquier dato que el padre escriba en worker.stdin se convierte en legible en process.stdin en el trabajador.
- Por defecto, los flujos process.stdout y process.stderr en el trabajador son simplemente canalizados a los flujos correspondientes en el hilo padre. Esto significa, por ejemplo, que console.log() y console.error() producen una salida de la misma manera en un hilo de trabajo que en el hilo principal. Puedes anular este valor por defecto pasando stdout:true o stderr:true en el segundo argumento del constructor de Worker(). Si haces esto, entonces cualquier salida que el trabajador escriba en esos flujos será leída por el hilo principal en worker.stdout y worker.stderr hilos. (Aquí hay una inversión potencialmente confusa de las direcciones de los flujos, y ya vimos lo mismo con los procesos hijos anteriormente en el capítulo: los flujos de salida de un hilo trabajador son flujos de entrada para el hilo padre, y el flujo de entrada de un trabajador es un flujo de salida para el padre).
- Si un hilo de trabajo llama a process.exit(), sólo el hilo sale, no todo el proceso.

- Los hilos de trabajo no pueden cambiar el estado compartido del proceso del que forman parte. Funciones como `process.chdir()` y `process.setuid()` lanzarán excepciones cuando se invoquen desde un trabajador.
- Las señales del sistema operativo (como `SIGINT` y `SIGTERM`) sólo se envían al hilo principal; no pueden recibirse ni manejarse en los hilos trabajadores.

### 16.11.3 Canales de comunicación y MessagePorts

Cuando se crea un nuevo hilo worker, se crea junto con él un canal de comunicación que permite pasar mensajes de ida y vuelta entre el worker y el hilo padre. Como hemos visto, el hilo worker utiliza `threads.parentPort` para enviar y recibir mensajes hacia y desde el hilo padre, y el hilo padre utiliza el objeto `Worker` para enviar y recibir mensajes hacia y desde el hilo worker.

La API de hilos de trabajo también permite la creación de canales de comunicación personalizados utilizando la API `MessageChannel` definida por los navegadores web y cubierta en [§15.13.5](#). Si has leído esa sección, mucho de lo que sigue te resultará familiar.

Supongamos que un worker necesita manejar dos tipos diferentes de mensajes enviados por dos módulos diferentes en el hilo principal. Estos dos módulos diferentes podrían compartir el canal por defecto y enviar mensajes con `worker.postMessage()`, pero sería más limpio si cada módulo tiene su propio canal privado para enviar mensajes al worker. O consideremos el caso en el que el hilo principal crea dos workers independientes. Un canal de comunicación personalizado puede permitir que los dos workers

se comuniquen directamente entre sí en lugar de tener que enviar todos sus mensajes a través del hilo principal.

Crea un nuevo canal de mensajes con el constructor `MessageChannel()`. Un objeto `MessageChannel` tiene dos propiedades, llamadas `puerto1` y `puerto2`. Estas propiedades se refieren a un par de objetos `MessagePort`. Llamar a `postMessage()` en uno de los puertos hará que se genere un evento "mensaje" en el otro con un clon estructurado del objeto `Message`:

```
const threads = require("worker_threads"); let channel = new threads.  
MessageChannel(); channel.port2.on("message", console.log); // Registrar  
cualquier mensaje que recibamos channel.port1.postMessage("hello"); //  
Hará que se imprima "hello"
```

También se puede llamar a `close()` en cualquiera de los puertos para romper la conexión entre los dos puertos y señalar que no se intercambiarán más mensajes. Cuando se llama a `close()` en cualquiera de los puertos, se envía un evento de "cierre" a ambos puertos.

Observe que el ejemplo de código anterior crea un par de objetos `MessagePort` y luego utiliza esos objetos para transmitir un mensaje dentro del hilo principal. Para utilizar canales de comunicación personalizados con trabajadores, debemos transferir uno de los dos puertos desde el hilo en el que se crea al hilo en el que se utilizará. La siguiente sección explica cómo hacerlo.

#### 16.11.4. Transferencia de MessagePorts y Arrays tipificados

La función postMessage() utiliza el algoritmo de clonación estructurada, y como hemos señalado, no puede copiar objetos como SSockets y Streams. Puede manejar objetos MessagePort, pero sólo como un caso especial usando una técnica especial. El método postMessage() (de un objeto Worker, de threads.parentPort, o de cualquier objeto MessagePort) toma un segundo argumento opcional. Este argumento (llamado transferList) es un array de objetos que deben ser transferidos entre hilos en lugar de ser copiados.

Un objeto MessagePort no puede ser copiado por el algoritmo de clonación estructurada, pero puede ser transferido. Si el primer argumento de postMessage() ha incluido uno o más MessagePorts (anidados a una profundidad arbitraria dentro del objeto Message), entonces esos objetos MessagePort deben aparecer también como miembros del array pasado como segundo argumento. Hacer esto le dice a Node que no necesita hacer una copia del MessagePort, y en su lugar puede dar el objeto existente al otro hilo. La clave para entender, sin embargo, sobre la transferencia de valores entre hilos es que una vez que un valor es transferido, ya no puede ser utilizado en el hilo que llamó a postMessage().

Así es como se puede crear un nuevo MessageChannel y transferir uno de sus MessagePorts a un trabajador:

```
// Crear un canal de comunicación personalizado const threads =
require("worker_threads"); let channel = new threads.
MessageChannel();

// Utilizar el canal por defecto del trabajador para transferir un extremo del nuevo
// canal al trabajador. Supongamos que cuando el trabajador recibe este
// mensaje comienza inmediatamente a escuchar los mensajes en

el nuevo canal. worker.postMessage({ command: "changeChannel", data:
canal. puerto1 }, [ canal. puerto1 ]);

// Ahora envía un mensaje al trabajador utilizando nuestro extremo del canal
personalizado. port2.postMessage("¿Puedes oírmeme ahora?");

// Y escuchar las respuestas del trabajador también canal. port2.on("message",
handleMessagesFromWorker);
```

Los objetos MessagePort no son los únicos que pueden ser transferidos. Si llama a postMessage() con un array tipado como mensaje (o con un mensaje que contenga uno o más arrays tipados anidados arbitrariamente dentro del mensaje), ese array tipado (o esos arrays tipados) serán simplemente copiados por el algoritmo de clonación estructurada. Pero las matrices tipificadas pueden ser grandes; por ejemplo, si está usando un hilo de trabajo para hacer el procesamiento de imágenes en millones de píxeles. Así que por eficiencia, postMessage() también nos da la opción de transferir arrays tipados en lugar de copiarlos. (Los hilos comparten la memoria por defecto. Los hilos de trabajo en JavaScript generalmente evitan la memoria compartida, pero cuando permitimos este tipo de transferencia controlada, se puede hacer muy eficientemente). Lo que hace que esto sea seguro es que cuando un array tipado se transfiera a otro hilo, se vuelve inutilizable en el hilo que lo transfirió. En el escenario de procesamiento de imágenes, el hilo principal podría transferir los

píxeles de una imagen al hilo trabajador, y luego el hilo trabajador podría transferir los píxeles procesados de vuelta al hilo principal cuando haya terminado. La memoria no necesitaría ser copiada, pero nunca sería accesible por dos hilos a la vez.

Para transferir una matriz tipificada en lugar de copiarla, incluya el ArrayBuffer que respalda la matriz en el segundo argumento de postMessage():

```
let pixels = new Uint32Array(1024*1024); // 4 megabytes de memoria  
// Supongamos que leemos algunos datos en este array tipado, y luego transferimos el  
// los píxeles a un trabajador sin copiarlos. Tenga en cuenta que no ponemos el propio  
array // en la lista de transferencia, sino el objeto Buffer del array. worker.  
postMessage(pixels, [ pixels.buffer ]);
```

Al igual que con los MessagePorts transferidos, un array tipado transferido queda inutilizado una vez transferido. No se lanza ninguna excepción si se intenta utilizar un MessagePort o un array tipado que ha sido transferido; estos objetos simplemente dejan de hacer algo cuando se interactúa con ellos.

### 16.11.5. Compartir arrays tipificados entre hilos

Además de transferir arrays tipados entre hilos, es posible compartir un array tipado entre hilos. Simplemente crea un SharedArrayBuffer del tamaño deseado y luego usa ese buffer para crear un array tipado. Cuando un array tipado que está respaldado por un

SharedArrayBuffer se pasa a través de postMessage(), la memoria subyacente será compartida entre los hilos. En este caso, no se debe incluir el buffer compartido en el segundo argumento de postMessage().

Sin embargo, no deberías hacer esto, porque JavaScript nunca fue diseñado con la seguridad de los hilos en mente y la programación multihilo es muy difícil de hacer bien. (Y esta es la razón por la que el SharedArrayBuffer no fue cubierto en §11.2: es una característica de nicho que es difícil de hacer bien). Incluso el simple operador `++` no es seguro para los hilos porque necesita leer un valor, incrementarlo y escribirlo de nuevo. Si dos hilos están incrementando un valor al mismo tiempo, a menudo sólo se incrementará una vez, como demuestra el siguiente código:

```

const threads = require("worker_threads");

if (threads.isMainThread) {
    // En el hilo principal, creamos un array de tipo compartido con
    // un elemento. Ambos hilos podrán leer y escribir // sharedArray[0] al mismo
    // tiempo.
    let sharedBuffer = new SharedArrayBuffer(4); let sharedArray = new
    Int32Array(sharedBuffer);

    // Ahora crea un hilo de trabajo, pasándole el array compartido con
    // como su valor inicial workerData para que no tengamos que molestarnos en //
    enviar y recibir un mensaje let worker = new threads.Worker(__filename, {
        workerData: sharedArray });

    // Esperar a que el trabajador comience a funcionar y luego incrementar
    // el entero compartido 10 millones de veces.
    worker.on("online", () => { for(let i = 0; i < 10_000_000; i++) sharedArray[0]++;
        // Una vez que hemos terminado con nuestros incrementos, empezamos a escuchar
        // los eventos de // mensajes para saber cuándo ha terminado el trabajador. worker.
        on("message", () => {
            // Aunque el entero compartido ha sido incrementado // 20 millones de veces, su
            // valor será generalmente mucho menor. // En mi ordenador el valor final suele ser
            // inferior a 12 millones.
            console.log(sharedArray[0]);
        });
    });

} si no {
    // En el hilo del trabajador, obtenemos el array compartido de workerData // y
    // lo incrementamos 10 millones de veces.
    let sharedArray = threads.workerData; for(let i = 0; i < 10_000_000; i++)
    sharedArray[0]++; // Cuando terminemos de incrementar, avisa a la threads principal.
    parentPort.postMessage("done");
}

```

Un escenario en el que podría ser razonable utilizar un SharedArrayBuffer es cuando los dos hilos operan en secciones completamente separadas de la memoria compartida. Podrías hacer esto creando dos matrices tipificadas que sirvan como vistas de regiones no superpuestas del buffer compartido, y luego hacer que tus dos hilos usen esas dos matrices tipificadas separadas. Una ordenación paralela podría hacerse así: un hilo ordena la mitad inferior de un array y el otro hilo ordena la mitad superior, por ejemplo. O algunos tipos de algoritmos de procesamiento de imágenes también son adecuados para este enfoque: múltiples hilos trabajando en regiones separadas de la imagen.

Si realmente debes permitir que varios hilos accedan a la misma región de un array compartido, puedes dar un paso hacia la seguridad de los hilos con las funciones definidas por el objeto Atomics. Atomics se añadió a JavaScript cuando SharedArrayBuffer para definir operaciones atómicas sobre los elementos de un array compartido. Por ejemplo, la función Atomics.add() lee el elemento especificado de una matriz compartida, le añade un valor especificado y escribe la suma de nuevo en la matriz. Lo hace de forma atómica, como si se tratara de una sola operación, y se asegura de que ningún otro hilo pueda leer o escribir el valor mientras se realiza la operación. Atomics.add() nos permite reescribir el código de incremento paralelo que acabamos de ver y obtener el resultado correcto de 20 millones de incrementos de un elemento del array compartido:

```

const threads = require("worker_threads");

if (threads.isMainThread) { let sharedBuffer = new SharedArrayBuffer(4); let
sharedArray = new Int32Array(sharedBuffer); let worker = new threads.
Worker(__filename, { workerData: sharedArray });

worker.on("online", () => {
  for(let i = 0; i < 10_000_000; i++) {
    Atomics.add(sharedArray, 0, 1); // Threadsafe atomic increment }

worker.on("mensaje", (mensaje) => {
  // Cuando los dos hilos hayan terminado, utiliza una función threadsafe
  // para leer el array compartido y confirmar que tiene el valor // esperado de
20.000.000.
  console.log(Atomics.load(sharedArray, 0)); });
}); } else { let sharedArray = threads.workerData; for(let
i = 0; i < 10_000_000; i++) {
  Atomics.add(sharedArray, 0, 1); // Threadsafe atomic increment } threads.
parentPort.postMessage("done");
}
}

```

Esta nueva versión del código imprime correctamente el número 20.000.000. Pero es unas nueve veces más lenta que el código incorrecto al que sustituye. Sería mucho más simple y mucho más rápido hacer los 20 millones de incrementos en un solo hilo. También hay que tener en cuenta que las operaciones atómicas pueden garantizar la seguridad de los hilos para los algoritmos de procesamiento de imágenes para los que cada elemento del array es un valor totalmente independiente de todos los demás valores. Pero en la mayoría de los programas del mundo real, varios

elementos del array suelen estar relacionados entre sí y se requiere algún tipo de sincronización de hilos de alto nivel. Las funciones de bajo nivel Atomics.wait() y

La función Atomics.notify() puede ayudar con esto, pero una discusión de su uso está fuera del alcance de este libro.

## 16.12 Resumen

Aunque JavaScript se creó para ejecutarse en navegadores web, Node ha convertido a JavaScript en un lenguaje de programación de uso general. Es especialmente popular para implementar servidores web, pero sus profundos vínculos con el sistema operativo significan que también es una buena alternativa a los scripts de shell.

Los temas más importantes que se tratan en este largo capítulo son:

- Las APIs asíncronas por defecto de Node y su estilo de concurrencia basado en un solo hilo, callback y eventos.
  - Tipos de datos fundamentales de Node, buffers y streams.
  - Los módulos "fs" y "path" de Node para trabajar con el sistema de archivos.
- Los módulos "http" y "https" de Node para escribir clientes y servidores HTTP.
- Módulo "net" de Node para escribir clientes y servidores no HTTP.
- El módulo "child\_process" de Node para crear y comunicarse con los procesos hijos.

- El módulo "worker\_threads" de Node para una verdadera programación multihilo utilizando el paso de mensajes en lugar de la memoria compartida.
- 

Node define una función `fs.copyFile()` que se utilizaría en la práctica.

<sup>1</sup>

A menudo es más limpio y sencillo definir el código del trabajador en un archivo separado. Pero este truco

<sup>2</sup> de tener dos hilos ejecutando diferentes secciones del mismo archivo me dejó boquiabierto cuando lo encontré por primera vez en la llamada al sistema de Unix `fork()`. Y creo que vale la pena demostrar esta técnica simplemente por su extraña elegancia.

# Capítulo 17. Herramientas y extensiones de JavaScript

---

Enhorabuena por haber llegado al último capítulo de este libro. Si has leído todo lo que viene antes, ahora tienes una comprensión detallada del lenguaje JavaScript y sabes cómo usarlo en Node y en los navegadores web. Este capítulo es una especie de regalo de graduación: introduce un puñado de importantes herramientas de programación que muchos programadores de JavaScript encuentran útiles, y también describe dos extensiones del núcleo del lenguaje JavaScript ampliamente utilizadas. Tanto si decides utilizar estas herramientas y extensiones para tus propios proyectos como si no, es casi seguro que las verás utilizadas en otros proyectos, por lo que es importante que al menos sepas lo que son.

Las herramientas y extensiones del lenguaje que se tratan en este capítulo son:

- ESLint para encontrar posibles errores y problemas de estilo en su código.
- Prettier para formatear su código JavaScript de manera estandarizada.
- Jest
  - como una solución todo en uno para escribir pruebas
  - unitarias de JavaScript.

npm para gestionar e instalar las bibliotecas de software de las que depende su programa.

- Herramientas de agrupación de código -como webpack, Rollup y Parcel- que convierten tus módulos de código JavaScript en un único paquete para su uso en la web.
- Babel para traducir el código JavaScript que utiliza nuevas características del lenguaje (o que utiliza extensiones del lenguaje) en código JavaScript que puede ejecutarse en los navegadores web actuales.
- La extensión de lenguaje JSX (utilizada por el marco de trabajo React) que permite describir interfaces de usuario utilizando expresiones de JavaScript que se parecen al marcado HTML.
- La extensión del lenguaje Flow (o la extensión similar TypeScript) que le permite anotar su código JavaScript con tipos y comprobar su código para la seguridad de tipos.

Este capítulo no documenta estas herramientas y extensiones de forma exhaustiva. El objetivo es simplemente explicarlas con la suficiente profundidad como para que puedas entender por qué son útiles y cuándo podrías querer utilizarlas. Todo lo que se trata en este capítulo se utiliza ampliamente en el mundo de la programación en JavaScript, y si decides adoptar una herramienta o extensión, encontrarás mucha documentación y tutoriales en línea.

## 17.1. La limpieza con ESLint

En programación, el término "lint" se refiere al código que, aunque es técnicamente correcto, es antiestético, o un posible error, o subóptimo de alguna manera. Un *linter* es una herramienta para detectar pelusas en tu código, y *linting* es el proceso de ejecutar un linter en tu código (y luego arreglar tu código para eliminar las pelusas para que el linter ya no se queje).

El linter más utilizado hoy en día para JavaScript es [ESLint](#). Si lo ejecuta y luego se toma el tiempo de arreglar los problemas que señala, hará que su código sea más limpio y menos probable que tenga errores. Considere el siguiente código:

```
var x = 'sin usar';

export function factorial(x) { if (x == 1) { return 1; } else { return x * factorial(x-1) }
}
```

Si ejecuta ESLint en este código, puede obtener una salida como la siguiente:

```
$ eslint code/ch17/linty.js

code/ch17/linty.js
1:1 error Var inesperado, utilice let o const en su lugar no-var
1:5 error 'x' tiene asignado un valor pero nunca se ha utilizado no-unused-vars
1:9 advertencia Las cadenas deben utilizar comillas dobles
4:11 error Se esperaba '===' y en su lugar se vio '==' eqequeq
Error 5:1 Se esperaba una sangría de 8 espacios pero se encontró una sangría de 6
7:28 error Falta el punto y coma
```

✖ 6 problemas (5 errores, 1 advertencia)  
3 errores y 1 advertencia potencialmente corregibles con la opción `--fix`.

Las cadenas pueden parecer a veces puntuosas. ¿Realmente importa si usamos comillas dobles o simples para nuestras cadenas? Por otro lado, la indentación correcta es importante para la legibilidad, y el uso de === y let en lugar de == y var te protege de errores sutiles. Y las variables que no se utilizan son un peso muerto en el código, no hay razón para mantenerlas.

ESLint define muchas reglas de linting y tiene un ecosistema de plug-ins que añaden muchas más. Pero ESLint es totalmente configurable, y puedes definir un archivo de configuración que

ajuste ESLint para que aplique exactamente las reglas que quieras y sólo esas reglas.

## 17.2 Formato de JavaScript con Prettier

Una de las razones por las que algunos proyectos utilizan linters es para imponer un estilo de codificación consistente, de modo que cuando un equipo de programadores trabaje en una base de código compartida, utilice convenciones de código compatibles. Esto incluye reglas de indentación del código, pero también puede incluir cosas como qué tipo de comillas se prefieren y si debe haber un espacio entre la palabra clave `for` y el paréntesis abierto que la sigue.

Una alternativa moderna a la aplicación de reglas de formato de código a través de un linter es adoptar una herramienta como [Prettier](#) para analizar y reformatear automáticamente todo su código.

Suponga que ha escrito la siguiente función, que funciona, pero con un formato poco convencional:

```
function factorial(x) { if(x==1){return 1} else{return  
x*factorial(x-1)} }
```

Al ejecutar Prettier en este código se corrige la sangría, se añaden los puntos y comas que faltan, se añaden espacios alrededor de los operadores binarios y se insertan saltos de línea después de `{` y antes de `}`, lo que da como resultado un código de aspecto mucho más convencional:

```
$ prettier factorial.js function
factorial(x) { if (x === 1) { return
1;
} si no {
    devuelve x * factorial(x - 1);
}
}
```

Si invoca a Prettier con la opción --write, simplemente reformateará el archivo especificado en su lugar en lugar de imprimir una versión reformateada. Si utiliza git para gestionar su código fuente, puede invocar Prettier con la opción --write en un hook de confirmación para que el código se formatee automáticamente antes de ser registrado.

Prettier es especialmente potente si configuras tu editor de código para que lo ejecute automáticamente cada vez que guardes un archivo. Me parece liberador escribir código descuidado y ver que se arregla automáticamente por mí.

Prettier es configurable, pero sólo tiene unas pocas opciones. Puede seleccionar la longitud máxima de la línea, la cantidad de sangría, si se debe usar el punto y coma, si las cadenas deben estar entre comillas simples o dobles, y algunas otras cosas. En general, las opciones por defecto de Prettier son bastante razonables. La idea es que simplemente adopte Prettier para su proyecto y no tenga que pensar nunca más en el formato del código.

Personalmente, me gusta mucho usar Prettier en los proyectos de JavaScript. Sin embargo, no lo he utilizado para el código de este libro, porque en gran parte de mi código dependo de un cuidadoso formateo manual para alinear mis comentarios verticalmente, y Prettier los desordena.

## 17.3 Pruebas unitarias con Jest

Escribir pruebas es una parte importante de cualquier proyecto de programación no trivial. Los lenguajes dinámicos como JavaScript soportan marcos de pruebas que reducen drásticamente el esfuerzo requerido para escribir pruebas, ¡y casi hacen que escribir pruebas sea divertido! Hay muchas herramientas y librerías de pruebas para JavaScript, y muchas de ellas están escritas de forma modular, de modo que es posible elegir una librería como corredor de pruebas, otra librería para las aserciones y una tercera para el mocking. En esta sección, sin embargo, vamos a describir [Jest](#), que es un marco popular que incluye todo lo que necesitas en un solo paquete.

Suponga que ha escrito la siguiente función:

```
const getJSON = require("./getJSON.js");

/**
 *      getTemperature() toma el nombre de una ciudad como entrada, y devuelve
 *      una Promesa que se resolverá a la temperatura actual de esa ciudad,
 *      en grados Fahrenheit. Se basa en un servicio web (falso) que devuelve
 *      temperaturas mundiales en grados Celsius.
 */
module.exports = async function getTemperature(city) { // Obtener la
temperatura en Celsius del servicio web let c = await getJSON(
`https://globaltemps.example.com/api/city/${ciudad}.toLowerCase
`);

// Convertir a Fahrenheit y devolver ese valor.
return (c * 5 / 9) + 32; // TODO: comprobarlo dos veces
}
```

()

fórmula };

Un buen conjunto de pruebas para esta función podría verificar que getTemperature() está obteniendo la URL correcta, y que está convirtiendo las escalas de temperatura correctamente. Podemos hacer esto con una prueba basada en Jest como la siguiente. Este código define una implementación falsa de getJSON() para que la prueba no haga realmente una petición de red. Y debido a que getTemperature() es una función asíncrona, las pruebas también son asíncronas; puede ser complicado probar funciones asíncronas, pero Jest lo hace relativamente fácil:

```

// Importar la función que vamos a probar const getTemperature =
require("./getTemperature.js");

// Y mock el módulo getJSON() del que depende getTemperature() jest.
mock("./getJSON"); const getJSON = require("./getJSON.js");

// Dile a la función mock getJSON() que devuelva una Promise ya resuelta // con
el valor de cumplimiento 0. getJSON.mockResolvedValue(0);

// Nuestro conjunto de pruebas para getTemperature() comienza aquí
describe("getTemperature()", () => {
  // Esta es la primera prueba. Nos aseguramos de que getTemperature()
llame // a getJSON() con la URL que esperamos test("Invoca la API correcta",
  async () => { let expectedURL =
"https://globaltemps.example.com/api/city/vancouver"; let t =
await(getTemperature("Vancouver"));

  // Los mocks de Jest recuerdan cómo fueron llamados, y podemos comprobarlo.
  expect(getJSON).toHaveBeenCalledWith(expectedURL);
});

// Esta segunda prueba verifica que getTemperature() convierte // Celsius a
Fahrenheit correctamente test("Convierte C a F correctamente", async () => {
  getJSON.mockResolvedValue(0); // Si getJSON devuelve 0C expect(await
  getTemperature("x")).toBe(32); // Esperamos 32F

  // 100C debería convertirse en 212F getJSON.mockResolvedValue(100); // Si
  getJSON.devuelve 100C expect(await getTemperature("x")).toBe(212); //
Esperamos 212F );
});

```

Con la prueba escrita, podemos utilizar el comando jest para ejecutarla, y descubrimos que una de nuestras pruebas falla:

```

$ jest getTemperature
FAIL ch17/getTemperature.test.js getTemperature()
  ✓ Invoca la API correcta (4ms)
  ✗ Convierte C a F correctamente (3ms)

  ● getTemperature() ' Convierte C a F correctamente expect(received).toBe(expected)
    // Object.is equality

      Previsto: 212
      Recibido: 87.55555555555556

29    | // 100C debe convertirse en 212F
30    | getJSON.mockResolvedValue(100); // Si getJSON devuelve 100C
> 31 | expect(await
getTemperature("x")).toBe(212); // Esperar 212F
      | ^
32    |   });
33    | });
34    | en Object.<anonymous>

```

(ch17/getTemperature.test.js:31:43)

Pruebas de suites: 1 fallida, 1 en total  
 Pruebas: 1 fallado, 1 aprobado, 2 en total  
 Instantáneas: 0 en total  
 Tiempo: 1.403s Se han realizado todas las pruebas que coinciden con /getTemperature/i.

Nuestra implementación de `getTemperature()` está utilizando una fórmula incorrecta para convertir C a F. Multiplica por 5 y divide por 9 en lugar de multiplicar por 9 y dividir por 5. Si corregimos el código y ejecutamos Jest de nuevo, podemos ver que las pruebas pasan. Y, como extra, si añadimos el argumento `-coverage` cuando invocamos a `jest`, calculará y mostrará la cobertura del código para nuestras pruebas:

```
$ jest --coverage getTemperature
PASS ch17/getTemperature.test.js getTemperature()
  ✓ Invoca la API correcta (3ms)
  ✓ Convierte correctamente de C a F (1ms)

-----|-----|-----|-----|---|
-----| File | % Stmts| % Branch| % Funcs| % Lines|
Línea descubierta #s|
-----|-----|-----|-----|---|
-----| Todos los archivos | 71.43| 100| 33.33| 83.33|
|
getJSON.js | 33.33| 100| 0| 50|
2|
getTemperature.js| 100| 100| 100|
|
-----|-----|-----|-----|---|
-----|
Suites de prueba: 1 aprobada, 1 en total
Pruebas: 2 aprobadas, 2 en total
Instantáneas: 0 en total
Tiempo: 1.508s Se han realizado todas las pruebas que coinciden
con /getTemperature/i.
```

La ejecución de nuestra prueba nos dio una cobertura de código del 100% para el módulo que estábamos probando, que es exactamente lo que queríamos. Sólo nos ha dado una cobertura parcial de `getJSON()`, pero hemos burlado ese módulo y no estábamos tratando de probarlo, así que es de esperar.

## 17.4 Gestión de paquetes con npm

En el desarrollo de software moderno, es común que cualquier programa no trivial que escribas dependa de bibliotecas de software de terceros. Si estás escribiendo un servidor web en Node, por ejemplo, podrías utilizar el framework Express. Y si estás creando una interfaz de usuario para ser mostrada en un navegador web, podrías usar un framework frontend como React o LitElement o Angular. Un gestor de paquetes facilita la

búsqueda e instalación de paquetes de terceros como estos. Igual de importante es que un gestor de paquetes hace un seguimiento de los paquetes de los que depende tu código y guarda esta información en un archivo para que cuando alguien más quiera probar tu programa, pueda descargar tu código y tu lista de dependencias, y luego usar su propio gestor de paquetes para instalar todos los paquetes de terceros que tu código necesita.

npm es el gestor de paquetes que se incluye con Node, y fue introducido en [§16.1.5](#). Sin embargo, es tan útil para la programación JavaScript del lado del cliente como para la programación del lado del servidor con Node.

Si estás probando el proyecto JavaScript de otra persona, entonces una de las primeras cosas que harás después de descargar su código es escribir npm install. Esto lee las dependencias listadas en el archivo *package.json* y descarga los paquetes de terceros que el proyecto necesita y los guarda en un directorio *node\_modules/*.

También puedes escribir npm install <nombre-del-paquete> para instalar un paquete concreto en el directorio *node\_modules/* de tu proyecto:

```
$ npm install express
```

Además de instalar el paquete nombrado, npm también hace un registro de la dependencia en el archivo *package.json* para el proyecto. Registrar las dependencias de esta manera es lo que permite a otros instalar esas dependencias simplemente escribiendo npm install.

El otro tipo de dependencia es el de las herramientas de desarrollo que necesitan los desarrolladores que quieren trabajar en su proyecto, pero que no son realmente necesarias para ejecutar el código. Si un proyecto utiliza Prettier, por ejemplo, para asegurar que todo su código tiene un formato consistente, entonces Prettier es una "dependencia dev", y puede instalar y registrar una de ellas con -save-dev:

```
$ npm install --save-dev prettier
```

A veces puedes querer instalar las herramientas de desarrollo de forma global para que sean accesibles en cualquier lugar, incluso para el código que no forma parte de un proyecto formal con un archivo *package.json* y un directorio *node\_modules/*. Para ello puedes utilizar la opción -g (para global):

```
$ npm install -g eslint jest
/usr/local/bin/eslint ->
/usr/local/lib/node_modules/eslint/bin/eslint.js
/usr/local/bin/jest ->
/usr/local/lib/node_modules/jest/bin/jest.js
+ jest@24.9.0
+ eslint@6.7.2
añadió 653 paquetes de 414 contribuyentes en 25,596s
```

```
$ que eslint
/usr/local/bin/eslint
$ que bromea
/usr/local/bin/jest
```

Además del comando "install", npm soporta los comandos "uninstall" y "update", que hacen lo que sus nombres dicen. npm también tiene un interesante comando "audit" que puedes usar para encontrar y arreglar vulnerabilidades de seguridad en tus dependencias:

```
$ npm audit --fix === npm audit security report ===
```

```
encontró 0 vulnerabilidades en  
876354 paquetes escaneados
```

Cuando instalas una herramienta como ESLint localmente para un proyecto, el script eslint termina en

`./node_modules/.bin/eslint`, lo que hace que el comando sea incómodo de ejecutar. Afortunadamente, npm viene con un comando conocido como "npx", que se puede utilizar para ejecutar las herramientas instaladas localmente con comandos como npx eslint o npx jest. (Y si usted utilice npx para invocar una herramienta que aún no ha sido instalada, la instalará por usted).

La compañía detrás de npm también mantiene el repositorio de paquetes <https://npmjs.com>, que contiene cientos de miles de paquetes de código abierto. Pero no es necesario utilizar el gestor de paquetes npm para acceder a este repositorio de paquetes. Las alternativas son [yarn](#) y [pnpm](#).

## 17.5 Agrupación de códigos

Si estás escribiendo un gran programa de JavaScript para ejecutarlo en los navegadores web, probablemente querrás utilizar una herramienta de agrupación de código, especialmente si utilizas bibliotecas externas que se entregan como módulos. Los desarrolladores web han estado utilizando módulos ES6 ([§10.3](#)) durante años, desde mucho antes de que las palabras clave de importación y exportación fueran soportadas en la web. Para ello, los programadores utilizan una herramienta de agrupación de

código que comienza en el punto de entrada principal (o puntos de entrada) del programa y sigue el árbol de directivas de importación para encontrar todos los módulos de los que depende el programa. A continuación, combina todos esos archivos de módulos individuales en un único paquete de código JavaScript y reescribe las directivas de importación y exportación para que el código funcione en esta nueva forma. El resultado es un único archivo de código que puede cargarse en un navegador web que no admite módulos.

Los módulos ES6 son soportados casi universalmente por los navegadores web hoy en día, pero los desarrolladores web todavía tienden a utilizar paquetes de código, al menos cuando lanzan código de producción. Los desarrolladores consideran que la experiencia del usuario es mejor cuando se carga un único paquete de código de tamaño medio cuando un usuario visita por primera vez un sitio web que cuando se cargan muchos módulos pequeños.

#### NOTA

El rendimiento de la web es un tema notoriamente complicado y hay muchas variables a tener en cuenta, incluyendo las continuas mejoras de los proveedores de navegadores, por lo que la única manera de estar seguro de la forma más rápida de cargar su código es probando a fondo y midiendo cuidadosamente. Ten en cuenta que hay una variable que está completamente bajo tu control: el tamaño del código. Menos código JavaScript siempre se cargará y ejecutará más rápido que más código JavaScript.

Hay un número de buenas herramientas de bundler de JavaScript disponibles. Los bundlers más utilizados son [webpack](#), [Rollup](#) y [Parcel](#). Las características básicas de los bundlers son más o menos

las mismas, y se diferencian en función de lo configurables que son o de lo fáciles que son de usar. Webpack ha existido durante mucho tiempo, tiene un gran ecosistema de plug-ins, es altamente configurable y puede soportar librerías no modulares más antiguas. Pero también puede ser complejo y difícil de configurar. En el otro extremo del espectro está Parcel, que pretende ser una alternativa de zeroconfiguración que simplemente hace lo correcto.

Además de realizar la agrupación básica, las herramientas de agrupación también pueden proporcionar algunas funciones adicionales:

- Algunos programas tienen más de un punto de entrada. Una aplicación web con múltiples páginas, por ejemplo, podría escribirse con un punto de entrada diferente para cada página. Los agrupadores generalmente permiten crear un paquete por punto de entrada o crear un único paquete que soporte múltiples puntos de entrada.
- Los programas pueden utilizar `import()` en su forma funcional ([§10.3.6](#)) en lugar de su forma estática para cargar dinámicamente los módulos cuando realmente se necesitan en lugar de cargarlos estáticamente al iniciar el programa. Hacer esto es a menudo una buena manera de mejorar el tiempo de inicio de su programa. Las herramientas Bundler que soportan `import()` pueden producir múltiples paquetes de salida: uno para cargar en el momento de inicio, y uno o más que se cargan dinámicamente cuando se necesitan. Esto puede funcionar bien si sólo hay unas pocas llamadas a `import()` en su programa y cargan módulos con conjuntos de dependencias relativamente disjuntos. Si los módulos cargados dinámicamente comparten dependencias, entonces se vuelve difícil averiguar cuántos

paquetes producir, y es probable que tenga que configurar manualmente su bundler para resolver esto.

- Por lo general, los empaquetadores pueden generar un archivo de *mapa de fuentes* que define una correspondencia entre las líneas de código en el paquete y las líneas correspondientes en los archivos fuente originales. Esto permite que las herramientas de desarrollo del navegador muestren automáticamente los errores de JavaScript en sus ubicaciones originales no empaquetadas.
- A veces, cuando importas un módulo a tu programa, sólo utilizas algunas de sus funciones. Una buena herramienta de bundler puede analizar el código para determinar qué partes no se utilizan y pueden omitirse de los bundles. Esta función recibe el caprichoso nombre de "agitarse el árbol".
- Los agrupadores suelen tener una arquitectura basada en plugins y admiten complementos que permiten importar y agrupar "módulos" que no son realmente archivos de código JavaScript. Suponga que su programa incluye una gran estructura de datos compatible con JSON. Los agrupadores de código pueden configurarse para permitirle mover esa estructura de datos a un archivo JSON separado y luego importarlo a su programa con una declaración como `import widgets from "./big-widget-list.json"`. Del mismo modo, los desarrolladores web que incrustan CSS en sus programas de JavaScript pueden utilizar los complementos de bundler que les permiten importar archivos CSS con una directiva de importación. Tenga en cuenta, sin embargo, que si importa cualquier cosa que no sea un archivo JavaScript, está utilizando una extensión JavaScript no estándar y haciendo que su código dependa de la herramienta bundler.
- En un lenguaje como JavaScript que no requiere compilación, la ejecución de una herramienta de bundler parece un paso de compilación, y es frustrante tener que ejecutar un bundler después de cada edición de código antes de poder ejecutar el código en el navegador. Los bundlers suelen admitir vigilantes del sistema de archivos que detectan las ediciones de cualquier

archivo en el directorio de un proyecto y regeneran automáticamente los bundles necesarios. Con esta función, normalmente puedes guardar el código y volver a cargar inmediatamente la ventana del navegador para probarlo.

- Algunos bundlers también admiten un modo de "sustitución de módulos en caliente" para los desarrolladores, en el que cada vez que se regenera un bundle, éste se carga automáticamente en el navegador. Cuando esto funciona, es una experiencia mágica para los desarrolladores, pero hay algunos trucos bajo el capó para que funcione, y no es adecuado para todos los proyectos.

## 17.6 Transpilación con Babel

Babel es una herramienta que compila JavaScript escrito con características del lenguaje moderno en JavaScript que no utiliza esas características del lenguaje moderno. Dado que compila JavaScript a JavaScript, a Babel se le llama a veces "transpilador". Babel se creó para que los desarrolladores web pudieran utilizar las nuevas características del lenguaje ES6 y posteriores sin dejar de dirigirse a los navegadores web que sólo admitían ES5.

Características del lenguaje como el operador de exponentiación `**` y las funciones de flecha pueden transformarse con relativa facilidad en `Math.pow()` y expresiones de función. Otras características del lenguaje, como la palabra clave `class`, requieren transformaciones mucho más complejas y, en general, la salida de código de Babel no está pensada para ser legible por los humanos. Sin embargo, al igual que las herramientas de bundler, Babel puede producir mapas de fuentes que mapean las ubicaciones del código transformado de vuelta a sus ubicaciones de fuente

originales, y esto ayuda dramáticamente cuando se trabaja con código transformado.

Los proveedores de navegadores están haciendo un mejor trabajo para mantenerse al día con la evolución del lenguaje JavaScript, y hoy en día hay mucha menos necesidad de compilar las funciones de flecha y las declaraciones de clase. Babel todavía puede ayudar cuando se quiere utilizar las últimas características como los separadores de guiones bajos en los literales numéricos.

Como la mayoría de las otras herramientas descritas en este capítulo, puedes instalar Babel con npm y ejecutarlo con npx.

Babel lee un archivo de configuración *.babelrc* que le indica cómo quiere que se transforme su código JavaScript. Babel define "preajustes" entre los que puedes elegir dependiendo de las extensiones del lenguaje que quieras utilizar y de la agresividad con la que quieras transformar las características estándar del lenguaje. Uno de los

Los preajustes interesantes de Babel son para la compresión de código por minificación

(eliminando los comentarios y los espacios en blanco, renombrando las variables, etc.).

Si utilizas Babel y una herramienta de agrupación de código, es posible que puedas configurar el agrupador de código para que ejecute automáticamente Babel en tus archivos JavaScript mientras construye el paquete por ti. Si es así, esta puede ser una opción conveniente porque simplifica el proceso de producción de código ejecutable. Webpack, por ejemplo, admite un módulo

"babel-loader" que se puede instalar y configurar para que ejecute Babel en cada módulo de JavaScript a medida que se agrupa.

Aunque hoy en día hay menos necesidad de transformar el núcleo del lenguaje JavaScript, Babel todavía se utiliza comúnmente para soportar extensiones no estándar del lenguaje, y describiremos dos de estas extensiones del lenguaje en las secciones siguientes.

## 17.7 JSX: Expresiones de marcado en JavaScript

JSX es una extensión del núcleo de JavaScript que utiliza una sintaxis de estilo HTML para definir un árbol de elementos. JSX se asocia más estrechamente con el marco React para interfaces de usuario en la web. En React, los árboles de elementos definidos con JSX se renderizan finalmente en un navegador web como HTML. Incluso si no tienes planes de utilizar React, su popularidad significa que es probable que veas código que utiliza JSX. Esta sección explica lo que necesitas saber para darle sentido. (Esta sección es sobre la extensión del lenguaje JSX, no sobre React, y sólo explica lo suficiente de React para proporcionar contexto para la sintaxis de JSX).

Puede pensar en un elemento JSX como un nuevo tipo de sintaxis de expresión de JavaScript. Los literales de cadena de JavaScript se delimitan con comillas, y los literales de expresiones regulares se delimitan con barras inclinadas. Del mismo modo, los literales de expresiones JSX se delimitan con corchetes. He aquí una muy sencilla:

```
let line = <hr>;
```

Si utiliza JSX, tendrá que utilizar Babel (o una herramienta similar) para compilar las expresiones JSX en JavaScript regular. La transformación es lo suficientemente sencilla como para que algunos desarrolladores opten por utilizar React sin usar JSX. Babel transforma la expresión JSX en esta declaración de asignación en una simple llamada a una función:

```
let line = React.createElement("hr", null);
```

La sintaxis de JSX es similar a la de HTML, y al igual que los elementos de HTML, los elementos de React pueden tener atributos como estos: `let image = < img src="logo.png" alt="The JSX logo" hidden>;`

Cuando un elemento tiene uno o más atributos, se convierten en propiedades de un objeto pasado como segundo argumento a `createElement()`:

```
let image = React.createElement("img", { src: "logo.png",  
alt: "El logo de JSX", hidden: true });
```

Al igual que los elementos HTML, los elementos JSX pueden tener cadenas y otros elementos como hijos. Al igual que los operadores aritméticos de JavaScript pueden utilizarse para escribir expresiones aritméticas de complejidad arbitraria, los elementos JSX también pueden anidarse a una profundidad arbitraria para crear árboles de elementos:

```
let sidebar = (  
< div className="sidebar">
```

```
< h1> Título</h1>
< hr/>
< p> Este es el contenido de la barra lateral</p>
</div>
);
```

Las expresiones regulares de llamadas a funciones de JavaScript también pueden anidarse a una profundidad arbitraria, y estas expresiones JSX anidadas se traducen en un conjunto de llamadas createElement() anidadas. Cuando un elemento JSX tiene hijos, esos hijos (que suelen ser cadenas y otros elementos JSX) se pasan como tercer argumento y siguientes:

```
let sidebar = React.createElement(
  "div", { className: "sidebar"}, // Esta llamada externa crea un <div> React.
  createElement("h1", null, // Este es el primer hijo del <div> "Título"), // y su propio
  primer hijo.  React.createElement("hr", null), // El segundo hijo del <div>.  React.
  createElement("p", null, // Y el tercer hijo.
    "Este es el contenido de la barra lateral"));

```

El valor devuelto por React.createElement() es un objeto JavaScript ordinario que es utilizado por React para renderizar la salida en una ventana del navegador. Como esta sección trata de la sintaxis JSX y no de React, no vamos a entrar en detalles sobre los objetos Element devueltos o el proceso de renderizado. Vale la pena señalar que puedes configurar Babel para compilar elementos JSX a invocaciones de una función diferente, por lo que si crees que la sintaxis JSX sería una forma útil de expresar otros tipos de estructuras de datos anidados, puedes adoptarla para tus propios usos no relacionados con React.

Una característica importante de la sintaxis JSX es que puede incrustar expresiones regulares de JavaScript dentro de expresiones JSX. Dentro de una expresión JSX, el texto entre llaves se interpreta como JavaScript plano. Estas expresiones anidadas

están permitidas como valores de atributos y como elementos hijos. Por ejemplo:

```
function sidebar(className, title, content, drawLine=true) { return (< div  
className={className}>  
  < h1> {título}</h1>  
  { drawLine && < hr/> }  
  < p> {contenido}</p>  
  </div>  
 );  
 }
```

La función sidebar() devuelve un elemento JSX. Toma cuatro argumentos que utiliza dentro del elemento JSX. La sintaxis de la llave rizada puede recordarle los literales de la plantilla que utilizan \${} para incluir expresiones de JavaScript dentro de las cadenas. Como sabemos que las expresiones JSX se compilan en invocaciones de funciones, no debería sorprender que se puedan incluir expresiones JavaScript arbitrarias porque las invocaciones de funciones también pueden escribirse con expresiones arbitrarias. Este código de ejemplo es traducido por Babel en lo siguiente:

```
function sidebar(className, title, content, drawLine=true) { return React.  
createElement("div", { className: className }, React.createElement("h1", null, title),  
  drawLine && React.createElement("hr", null), React.  
createElement("p", null, content));  
}
```

Este código es fácil de leer y entender: las llaves han desaparecido y el código resultante pasa los parámetros de la función entrante a React.createElement() de forma natural. Fíjate en el ingenioso truco que hemos hecho aquí con el parámetro drawLine y el operador && de cortocircuito. Si llamas a sidebar() con



Probando el código JavaScript, Explorando el orden de cotejo de JavaScript, Comparando colores de cadenas, Colores, patrones y degradados operador coma (,), El operador coma (,) comentarios sintaxis para, Un recorrido por JavaScript, Sintaxis de comentarios en ejemplos de código, Un recorrido por JavaScript método compare(), Comparación de cadenas operadores de comparación, Operadores de comparación composición, Translucidez y composición declaraciones compuestas, Declaraciones compuestas y vacías propiedades computadas, Nombres de propiedades computadas método concat(), Añadir arrays con concat() Operador de acceso condicional (?.), Un recorrido por JavaScript, Invocación condicional de funciones, Invocación condicional, Operador condicional de invocación de funciones (?:"), El operador condicional (?:) declaraciones condicionales, Declaraciones, Atributo configurable condicional, Introducción a los objetos, Atributos de propiedades Función console.log() de la API de la consola, La salida formateada de la API de la consola con, Salida

formateada con funciones de la consola definidas por,

La API de la consola-El soporte de la API de la consola

para, La función console.log() de la API de la consola,

Hola mundo, Palabra clave const de la salida de la

consola, Declaraciones con let y constantes

declarando, Visión general y definiciones,

Declaraciones con let y const, const, let y var

definición de término, Declaración de variables y

nomenclatura de asignaciones, Constructores de

palabras reservadas

Constructor Array(), El constructor Array() Constructor

Audio(), El constructor Audio() Clases y, Clases y

constructores-El constructor Propiedad, El constructor

Propiedad

constructores, identidad de la clase y instanceof,

Constructores, identidad de la clase y expresión instanceof

new.target, Clases y constructores invocación del

constructor, Invocación del constructor definición del

término, Funciones ejemplos de, Creación de objetos con

new

Constructor Function(), El constructor Function()

Constructor Set(), La clase Set

Encabezado HTTP Content-Security-Policy,

declaraciones Security continue, continue

estructuras de control, [Un recorrido por JavaScript](#)-[Un recorrido por JavaScript](#), [Declaraciones](#) [cookies](#)  
[API para manipular](#), [Definición del término](#) [cookies](#), [Atributos de vida](#) y alcance de las cookies, Atributos de las cookies:  
limitaciones de vida y alcance de, [Atributos de las cookies](#):  
origen del nombre de [vida y alcance](#), [Lectura de las cookies](#),  
[Almacenamiento de las cookies](#)  
transformaciones del sistema de [coordenadas](#),  
[Transformación del sistema de coordenadas](#)[Ejemplo de transformación](#) símbolo de copyright (\xA9), [Secuencias de escape en literales de cadena](#) método copyWithin(),  
[copyWithin\(\)](#)

API de gestión de credenciales, [criptografía](#) y APIs relacionadas  
Compartición de recursos entre orígenes (CORS), [La política del mismo origen](#), [Peticiones de origen cruzado](#) cross-site scripting (XSS), [Cross-site scripting](#)  
criptografía, [Números enteros de precisión arbitraria con BigInt](#), [Criptografía](#) y APIs relacionadas

Píxeles CSS, [coordinadas del documento](#) y [coordinadas de la ventana gráfica](#)

Hojas de estilo CSS estilos  
comunes CSS, [Scripting](#) estilos  
computados CSS, [Estilos](#)  
[computados](#)

[Animaciones](#) y [eventos](#) CSS, [Animaciones](#) y [eventos](#) CSS  
[Clases](#) CSS, [Clases](#) CSS

Sintaxis de los selectores CSS, [Selección de elementos con selectores CSS](#), Estilos en línea, Convenciones de nomenclatura de [los estilos en línea](#), Hojas de estilo de scripting, Objeto CSSStyleDeclaration de las hojas de estilo, Rizos de [los estilos en línea \({}\), Un recorrido por JavaScript](#), Moneda de los inicializadores de objetos y matrices, Curvas de [formato de los números, Curvas](#)

## D

propiedades de los datos, [Getters y Setters de la propiedad](#)

Clase DataView, [DataView y Endianness](#)

Tipo de fecha, [Visión general y definiciones](#),

[Fechas y horas aritmética de fechas](#), [Aritmética de fechas](#)

formatting and parsing date strings, [Formatting and Parsing Date Strings](#) formatting for internationalization, [Formatting Dates and Times](#)[Formatting Dates and Times](#) high-resolution timestamps, [Timestamps overview of](#), [Dates and Times](#), [Dates and Times timestamps](#) debugger statements, [debugger declarations class](#), [class const, let, and var, const, let, y var function](#), [function import and export](#), [import and export overview of](#), [Declaraciones decodeURI\(\) function](#), Legacy [URL Functions](#) decodeURIComponent() function, Legacy [URL Functions](#) decrement operator (--), [Unary Arithmetic Operators delegation](#), [Delegation Instead of Inheritance](#) delete operator, [The delete Operator](#), [Deleting Properties](#)

ataques de denegación de servicio, escritura en flujos y manejo de la contrapresión

asignación de desestructuración, asignación de desestructuración

Asignación, Desestructuración de Argumentos de Función

en ParámetrosDestructuración de Argumentos de Función

en Parámetros herramientas de desarrollo, Exploración del

evento devicemotion de JavaScript, APIs de dispositivos

móviles

evento de orientación del dispositivo, API para dispositivos móviles

propiedad devicePixelRatio, diccionarios de coordenadas de

documento y coordenadas de vista, introducción a los objetos,

objetos como matrices asociativas directorios (nodo), trabajo

con directorios función distance(), declaraciones de funciones

operador de división (/), aritmética en JavaScript, expresiones

aritméticas bucles do/while, do/while

geometría del documento y desplazamiento, Geometría del

documento y desplazamiento

Tamaño de la ventana, tamaño del contenido y posición de  
desplazamiento

Píxeles CSS, coordenadas del documento y coordenadas de la  
ventana gráfica

determinar el elemento en un punto, determinar el elemento en  
un punto

coordenadas del documento y coordenadas de la ventana gráfica,  
Coordenadas del documento y coordenadas de la ventana gráfica

Consulta de la geometría de los elementos, Consulta de la geometría de un elemento

Desplazamiento, Desplazamiento

Tamaño de la ventana gráfica, tamaño del contenido y posición de desplazamiento, Tamaño de la ventana gráfica, tamaño del contenido y posición de desplazamiento

Modelo de Objetos del Documento (DOM), El Modelo de Objetos del Documento-

Ejemplo: Generación de una estructura y un recorrido del documento del índice, Estructura y recorrido del documento

Nodos DocumentFragment, Uso de los componentes web generar dinámicamente tablas de contenido, Ejemplo: Generación de un índice de contenidos

Elementos iframe, Coordenadas del documento y coordenadas de la ventana gráfica modificación del contenido, Contenido de los elementos como HTML modificación de la estructura, Creación, inserción y eliminación de nodos visión general de, Documentos de scripting consulta y configuración de atributos, Atributos selección de elementos del documento, Selección de elementos del documento shadow DOM, Shadow DOM-Shadow DOM API DocumentFragment nodes, Uso de documentos de Web Components, carga de nuevos, Carga de nuevos documentos signo del dólar (\$), Identificadores y palabras reservadas

Evento DOMContentLoaded, [Ejecución de programas JavaScript](#), [Línea de tiempo del lado del cliente](#) operador de punto (.), [Un recorrido por JavaScript](#), [Consulta y configuración de propiedades](#) comillas dobles ("), [Literales de cadena](#) barras dobles (//), [Un recorrido por JavaScript](#), [Un recorrido por JavaScript](#), [Comentarios](#) función drawImage(), [Media APIs](#) operaciones de dibujo curvas, [Curvas](#) imágenes, [Imágenes](#) rectángulos, texto de [rectángulos](#), matrices dinámicas de [texto](#), [matrices](#)

## E

Norma ECMA402, [La API de internacionalización](#) ECMAScript (ES), [Introducción a JavaScript](#) método elementFromPoint(), coordenadas [del documento](#) y [coordenadas de la ventana gráfica](#) elementos de array definición del término, [Arrays](#) lectura y escritura, [Lectura y escritura de elementos de array](#) elementos personalizados, [Elementos personalizados](#)

determinar el elemento en un punto, determinar el elemento en un punto iframe, coordenadas del documento y coordenadas de la ventana gráfica

Consulta de la geometría de los elementos, Consulta de la geometría de un elemento seleccionando, Selección de elementos del documento

método `ellipse()`, Curvas declaraciones else if,

emojis else if, Unicode, Secuencias de escape

en literales de cadena declaraciones vacías,

Declaraciones compuestas y vacías cadenas

vacías, Función de texto `encodeURI()`,

Funciones de URL heredadas función

`encodeURIComponent()`, Funciones de URL

heredadas contracciones en inglés, Atributo

enumerable de literales de cadena,

Introducción a los objetos, Atributos de

propiedades operador de igualdad `(==)` visión

general de, Operadores de igualdad y

desigualdad

conversiones de tipo, Visión general y definiciones,

Conversiones y

Igualdad, Operadores de conversión de casos especiales

operadores de igualdad, Un recorrido por las clases de error de

JavaScript, Clases de error manejo de errores usando Promesas,

[Manejo de errores con Promesas](#), [Más sobre Promesas y Errores entorno del navegador web](#), [Errores del programa](#)

ES2016

operador de exponenciación (\*\*), [Aritmética en JavaScript](#),  
[Expresiones aritméticas](#) método includes(), [includes\(\)](#)

ES2017, palabras clave async y await, [El operador await](#),  
[Detalles de la implementación de JavaScript asíncrono](#), [async y await](#)

ES2018

iterador asíncrono, [iteración asíncrona con for/await](#), [iteración asíncrona](#)

desestructuración con parámetros de reposo, [Desestructuración de argumentos de función en parámetros](#)

Método .finally(), [Los métodos catch y finally expresiones regulares lookbehind](#) aserciones, [Especificación de posición de coincidencia](#) grupos de captura con nombre, [Alternancia, agrupación y referencias](#) s bandera, [Banderas](#)

Clases de caracteres Unicode, [Clases de caracteres](#)

[Operador de propagación \(...\)](#), [Operador de propagación](#),  
[Desestructuración de argumentos de función en parámetros](#),  
[Iteradores y generadores](#)

ES2019

Cláusulas catch desnudas, arrays de aplanamiento  
try/catch/finally, arrays de aplanamiento con flat() y  
flatMap()

ES2020

Operador "de primera" ("First-Defined")

Tipo BigInt, enteros de precisión arbitraria con BigInt

BigInt64Array(), Tipos de matrices tipificadas

BigUint64Array(), Tipos de matrices tipificadas

operador de acceso condicional (?.), Un recorrido por

JavaScript, Errores de acceso a propiedades, Invocación

condicional de funciones, Invocación condicional

globalThis, La función global object import(),

Importaciones dinámicas con import() lastIndex y

RegExp API, exec()

método matchAll(), matchAll(), exec(), Implementación de

objetos iterables precedencia del operador, Precedencia del

operador Promise.allSettled(), Promesas en expresiones de

acceso a propiedades en paralelo, Acceso a propiedades

condicional

ES5

método apply(), Los métodos call() y apply()

romper cadenas a través de múltiples líneas, literales de

cadena, secuencias de escape en los literales de cadena

errores resueltos por las variables de alcance de bloque,

línea de base de la compatibilidad de los cierres,

Introducción al método JavaScript Function.bind(), El

constructor Property getters and setters, Property Getters

and Setters IE11 workaround, Módulos de JavaScript en la

Web transpilación con Babel, Transpilación con Babel

## ES6

Función Array.of(), Funciones de flecha [Array.of\(\)](#),

[Definición de funciones](#), [Funciones de flecha](#) enteros

binarios y octales, [Literales enteros](#) función de

etiqueta incorporada, [Literales de plantilla](#)

[etiquetados](#)

declaración de clase, [clase](#)

[palabra clave class](#), [Clases con la palabra clave class-](#)

[Ejemplo: Una clase de números complejos](#) propiedades

computadas, [Nombres de propiedades computadas](#)

sintaxis literal de objetos extendida, [Propiedades](#)

[abreviadas bucles for/of, for/of/in](#)

IE11 workaround, [Módulos JavaScript en la Web](#) cadenas

iterables en, Arreglos iterables de [texto](#), [Arreglos iterables](#)

objeto matemático, [Aritmética en módulos JavaScript](#) en

importaciones dinámicas con import(), [Importaciones](#)

[dinámicas con import\(\)](#) exportaciones, [Exportaciones ES6](#)

import.meta.url, importaciones [import.meta.url](#),

[Importaciones ES6-ES6](#)

[importaciones y exportaciones con renombramiento](#),

[Importaciones y exportaciones con renombramiento](#)

Módulos de JavaScript en la web, Módulos de JavaScript en la

[web - Visión general de los módulos en ES6](#), [Reexportaciones](#)

de [módulos](#)

Promesas

Promesas encadenadas, [Promesas encadenadas-Promesas encadenadas](#)

manejo de errores con, [Más sobre Promesas y Errores-Los métodos catch y finally](#) haciendo Promesas, [Haciendo Promesas-Promesas en Secuencia](#) visión general de, [Promesas](#) operaciones paralelas, [Promesas en Paralelo](#)

[Promesas en secuencia](#), [Promesas en secuencia-Promesas en secuencia](#)

[Resolver las promesas](#), [Resolver las promesas-Más sobre las promesas y los errores](#)

retorno de las devoluciones de llamada de las promesas, [Los métodos catch y finally](#) utilizando, [Uso de las promesas-manejo de los errores con las promesas](#) orden de enumeración de las propiedades, [Orden de enumeración de las propiedades](#) liberación de, [Introducción a las clases Set y Map de JavaScript](#), [for/of](#) con métodos abreviados de [Set y Map](#), [Métodos abreviados operador de propagación \(...\)](#), [El operador de propagación](#) cadenas delimitadas con puntos suspensivos, [Literales de cadenas](#), [Literales de plantillas](#)

subclases con cláusula extends, [Subclases con extends y super-Subclases con extends y super](#) Tipo de símbolos, [Visión general y Definiciones](#) símbolos como nombres de propiedades, [Símbolos como nombres de propiedades](#) arrays tipados, [Arrays](#)

declaración de variables en, [Declaración de variables y asignación](#) palabra clave `yield*`, [yield\\*](#) y generadores recursivos secuencias de escape apóstrofes, [literales de cadena](#) en literales de cadena, [secuencias de escape en literales de cadena](#)

Unicode, [Secuencias de Escape Unicode](#) función `escape()`, [Funciones URL heredadas](#) ESLint, [Linting con ESLint](#) función `eval()`, Expresiones de [Evaluación-Estricta eval\(\)](#) global `eval()`, [Global eval\(\) estricta eval\(\)](#), [Expresiones de Evaluación-Estricta eval\(\)](#) escuchadores de eventos, [Eventos](#), [Eventos](#)

Modelo de programación orientado a eventos, [JavaScript asíncrono](#), [Eventos](#)

[Envío de eventos personalizados](#), [Eventos enviados por el servidor](#) definición del término, [JavaScript asíncrono](#) envío de eventos personalizados, [Envío de eventos personalizados](#) cancelación de eventos, [Cancelación de eventos](#) categorías de eventos, [Categorías de eventos](#) invocación de manejadores de eventos, [Invocación de manejadores de eventos](#) propagación de eventos, [Propagación de eventos](#) visión general de, [Eventos](#)

Registro de manejadores de eventos, [Registro de manejadores de eventos](#) enviados por el servidor,

Eventos enviados por el servidor características de la plataforma web a investigar, Eventos clase EventEmitter, Eventos y EventEmitter método every(), every() y some() excepciones, lanzar y atrapar, método throw exec(), exec() notación exponencial, Literales de punto flotante Operador de exponenciación (\*\*), Aritmética en JavaScript, Expresiones aritméticas declaración de exportación, importación y exportación palabra clave, Módulos en ES6 declaraciones de expresión, Expresiones expresiones aritméticas, Expresiones aritméticas-Operadores Bitwise expresiones de asignación, Expresiones de asignación-Asignación con definición de operación de término, Expresiones y operadores incrustados dentro de literales de cadena, Expresiones de evaluación de literales de cadena, Expresiones de evaluación-Evalu() estricto formando con operadores, Un paseo por JavaScript, Expresiones y operadores expresiones de definición de funciones, Expresiones de definición de funciones, Expresiones de funciones, Funciones como espacios de nombres expresión inicializadora, Un recorrido por JavaScript, Inicializadores de objetos y matrices expresiones de invocación, Expresiones de invocación-invocación condicional, Invocación de funciones-invocación de constructores expresiones lógicas, Expresiones lógicas-

NO (!) expresión new.target, Clases y constructores inicializadores de objetos y arrays, Inicializadores de objetos y arrays expresiones de creación de objetos, Expresiones de creación de objetos expresiones primarias, Expresiones primarias expresiones de acceso a la propiedad, Expresiones de acceso a la propiedad  
Acceso condicional a la propiedad, Expresiones relacionales, Expresiones relacionales-El operador instanceof frente a las declaraciones, Un recorrido por JavaScript, Extensibilidad de las declaraciones, Extensibilidad de los objetos

## F

Función factorial(), Declaraciones de funciones, Funciones de fábrica de invocación de funciones, Clases y prototipos, valores falsos, Función fetch() de valores booleanos, Método fetch() de eventos de red, abortar peticiones, Abortar una petición cross-origin requests, Cross-origin requests examples of, fetch() file upload, File upload with fetch()

Códigos de estado HTTP, cabeceras de respuesta y errores de red, Códigos de estado HTTP, cabeceras de respuesta y errores de red, opciones varias de la solicitud, Opciones varias de la solicitud, análisis de los cuerpos de la respuesta, Análisis de los

cuerpos de la respuesta configuración de las cabeceras de la solicitud, [Configuración de las cabeceras de la solicitud](#) configuración de los parámetros de la solicitud, [Configuración de los parámetros de la solicitud](#) especificando el método de solicitud y el cuerpo de la solicitud, [Especificando el método de solicitud y el cuerpo de la solicitud](#) pasos de, [fetch\(\) streaming de cuerpos de respuesta](#), [Streaming de cuerpos de respuesta](#) campos, públicos, privados y estáticos, [Public, Private, and Static Fields](#) file handling (Node), [Working with Files-Working with Directories](#) directories, [Working with Directories](#) file metadata, [File Metadata](#) file mode strings, [Writing Files](#) file operations, [File Operations](#) overview of, [Working with Files](#) rutas, [descriptores de archivos y FileHandles](#), [rutas, descriptores de archivos y FileHandles](#) lectura de archivos, [lectura de archivos](#) escritura de archivos método [fill\(\)](#), [fill\(\)](#) método [filter\(\)](#), [filter\(\)](#) Método [.finally\(\)](#), [Los métodos catch y finally-Los métodos catch y finally](#) números de cuentas financieras, [Almacenamiento](#) método [find\(\)](#), [find\(\)](#) y [findIndex\(\)](#) método [findIndex\(\)](#), [find\(\)](#) y [findIndex\(\)](#) Herramientas para desarrolladores de Firefox, [Explorando JavaScript](#) operador definido por primera vez (??), Método [flat\(\)](#) definido por primera vez (??) método [flatMap\(\)](#),

Aplanando matrices con flat() y flatMap() método flatMap(),

Aplanando matrices con flat() y flatMap()

Literales en coma flotante, Literales en coma flotante, errores de redondeo y coma flotante binario

Ampliación del lenguaje de flujo, comprobación de tipos con flujo numerado

Tipos y uniones

discriminadas tipos de

matrices, tipos de matrices

tipos de clases, tipos de

clases

Tipos enumerados y uniones discriminadas, Tipos enumerados y uniones discriminadas tipos de función, Tipos de función

instalación y ejecución, Instalación y ejecución del flujo

tipos de objeto, Tipos de objeto otros tipos

parametrizados, Otros tipos parametrizados visión

general de, Comprobación de tipos con Flow tipos de

sólo lectura, Tipos de sólo lectura alias de tipos, Alias

de tipos

TypeScript frente a Flow, Comprobación de tipos

con Flow tipos de unión, Tipos de unión usando

anotaciones de tipo, Uso de anotaciones de tipo

para bucles, for, Iteración de matrices

Bucles for/await, Iteración asíncrona con for/await, Iteración

asíncrona para bucles for/in, for/in, Enumeración de

propiedades

bucles for/of, Texto, for/of-for/in, Iteración de matrices,  
Iteradores y generadores método forEach(), Iteración de matrices, método forEach() format(), Formato de números fracciones, Formato de números método fromData(), Análisis de los cuerpos de respuesta front-end JavaScript, JavaScript en los navegadores web módulo fs (Node), Trabajar con archivos-  
Trabajar con directorios declaración de funciones, expresiones de funciones, Expresiones de funciones, Funciones como espacios de nombres palabra clave function, Definición de funciones

Constructor Function(), La palabra clave function\* del constructor Function(), Generadores funciones funciones flecha, Un recorrido por JavaScript, Definición de funciones, Funciones flecha sensibilidad a mayúsculas y minúsculas, El texto de un programa JavaScript cierres, Cierres-Cierres definiendo, Definición de funciones-Funciones anidadas definir sus propias propiedades de función, Definir sus propias propiedades de función funciones de fábrica, Clases y prototipos argumentos de función y parámetros tipos de argumentos, Tipos de argumentos objeto, El objeto Argumentos destructuring function arguments into parameters, Destructuring Function Arguments into ParametersDestructuring Function Arguments into Parameters

parámetros opcionales y valores por defecto, Parámetros opcionales y valores por defecto visión general de, Argumentos y parámetros de la función

Parámetros restantes, Parámetros restantes y listas de argumentos de longitud variable

operador de propagación para las llamadas de función, El operador de propagación para Llamadas de función

listas de argumentos de longitud variable, Rest Parameters and Variable-

Longitud de las listas de argumentos expresiones de definición

de funciones, Expresiones de definición de funciones

invocación de funciones, Creación de objetos con new

propiedades, métodos y constructor de funciones, Function

Propiedades, métodos y constructor-El constructor de Function()

método bind(), El método bind() métodos call() y apply(), Los

métodos call() y apply() constructor de Function(), El constructor

de Function() propiedad length, La propiedad length propiedad

name, La propiedad name propiedad prototype, La propiedad

prototype método toString(), El método toString() programación

funcional exploración, Programación funcional funciones de

orden superior, Funciones de orden superior memoización,

Memoización

aplicación parcial de funciones, Aplicación parcial de funciones

Procesamiento de matrices con función, Procesamiento de matrices con

Funciones como espacios de nombres, Funciones como espacios de nombres  
funciones como valores, Funciones como Valores-Definiendo su propio  
Propiedades de la función  
Invocación de enfoques, Invocación de funciones Invocación de constructores, Ejemplos de invocación de constructores,  
Recorrido por JavaScript Invocación de funciones implícitas,  
Invocación de funciones implícitas Invocación indirecta,  
Invocación indirecta Expresiones de invocación, Invocación de funciones Invocación de métodos  
Invocación de métodos, Palabras reservadas Visión general de,  
Visión general y definiciones, Funciones recursivas, Invocación de funciones Sintaxis abreviada de, Recorrido por JavaScript  
Funciones de matrices estáticas, Funciones de matrices estáticas

## G

Recogida de basura, visión general y definiciones  
Funciones generadoras, Generadores, El valor de retorno de una función generadora (véase también iteradores y generadores) API de geolocalización, API de dispositivos móviles  
Método getBoundingClientRect(), Coordenadas del documento y coordenadas de la ventana gráfica

Método getRandomValues(), Criptografía y APIs relacionadas

métodos getter, Getters y Setters de propiedades, Getters, Setters y otros

Formularios del método

global eval(), Objeto global eval(), El objeto global, El objeto global en los navegadores web variables globales, Degradados de alcance variable y constante, Gráficos de colores, patrones y degradados

3D, Gráficos en un <lienzo>

API de lienzo, Gráficos en un <lienzo> -Manipulación de píxeles dimensiones y coordenadas del lienzo, Recorte de dimensiones y coordenadas del lienzo, Recorte

transformaciones del sistema de coordenadas,

Transformación del sistema de coordenadas-Ejemplo

de operaciones de dibujo, Operaciones de dibujo en

el lienzo atributos de los gráficos, Atributos de los

gráficos visión general de, Gráficos en un <lienzo>

trayectorias y polígonos, Trayectos y polígonos

manipulación de píxeles, Manipulación de píxeles

guardar y restaurar el estado de los gráficos, Guardar y restaurar el estado de los gráficos

gráficos vectoriales escalables (SVG), SVG: Scalable Vector Graphics-

Creación de imágenes SVG con el

operador JavaScript mayor que (>)

visión general de, Operadores de comparación

Comparación de cadenas, Trabajar con conversiones de tipo Strings, Conversiones de operadores de casos especiales mayor o igual que el operador ( $\geq$ ) descripción general de,

Operadores de comparación comparación de cadenas, Trabajar con conversiones de tipo Strings, Conversiones de operadores de casos especiales

## H

Eventos hashchange, Gestión del Historial con Eventos hashchange hashtables, Introducción a los Objetos, Objetos como Arrays Asociativos operador hasOwnProperty, Probando Propiedades Hola Mundo, Hola Mundo, Salida de Consola

literales hexadecimales, literales enteros, secuencias de escape en literales de cadena funciones de orden superior, funciones de orden superior

histogramas, frecuencia de caracteres, Ejemplo: Histogramas de frecuencia de caracteres-Resumen método history.pushState(), Gestión del historial con el método pushState() método history.replaceState(), Gestión del historial con el levantamiento de pushState(), Declaraciones de variables con var

Etiquetas HTML <script>, JavaScript en etiquetas HTML  
<script>-Carga de scripts bajo demanda directivas de importación y exportación, Módulos de carga de scripts bajo demanda, Carga de scripts bajo demanda

especificando el tipo de script, Especificando el tipo de script  
Ejecución sincrónica de scripts, Cuando los scripts se ejecutan: propiedad de texto asíncrono y diferido, Contenido del elemento como texto plano

Etiqueta HTML <template>, Plantillas HTML

Código HTML, comillas simples y dobles en, Literales de cadena  
Clientes y servidores HTTP, Clientes y servidores HTTP-Clientes y servidores HTTP

I  
identificadores, sensibilidad a las mayúsculas y minúsculas, El texto de un programa JavaScript propósito de, identificadores y palabras reservadas, declaración y asignación de variables palabras reservadas, identificadores y palabras reservadas, sintaxis de las expresiones primarias, ideogramas de los identificadores y palabras reservadas, Unicode

Sentencias if, Sentencia if-if if/else, Valores booleanos imágenes dibujo en Canvas, Imágenes manipulación de píxeles, Manipulación de píxeles expresión de función invocada inmediatamente, Funciones como espacios de

nombres inmutabilidad, Trabajar con cadenas, Valores primitivos inmutables y referencias de objetos mutables invocación implícita de funciones, Invocación implícita de funciones declaración import, import y export palabra clave import, Módulos en ES6 función import(), Importaciones dinámicas con import() operador import.meta.url, import.meta.url operador in, El operador in, Comprobación de propiedades método includes(), includes() operador de incremento (++), Operadores aritméticos unarios posición del índice, Arrays, Lectura y escritura de elementos de arrays IndexedDB, Hilos de trabajo de IndexedDB y mensajería método indexOf(), indexOf() y lastIndexOf() invocación indirecta, Invocación indirecta operador de desigualdad (!==) valores booleanos, Visión general de los valores booleanos, Operadores de igualdad y desigualdad comparación de cadenas, Trabajar con cadenas valor infinito, Aritmética en JavaScript herencia, introducción a los objetos, herencia, delegación en lugar de herencia expresión inicializadora, Un recorrido por JavaScript, Objeto y Matriz Inicializadores métodos de instancia, métodos estáticos operador instanceof, El operador instanceof, Constructores, Identidad de clase y literales enteros

[instanceof](#), [Literales enteros](#) [clases de la API de internacionalización](#) incluidas en, [La API de internacionalización](#) [comparar cadenas](#), [Comparar cadenas-Comparar cadenas](#)

[Formato de fechas y horas](#), [Formato de fechas y horas](#)[Formato de fechas y horas](#)[Formato de números](#), [Formato de números](#)[Soporte de números](#) para en Node, [La API de internacionalización](#) texto traducido, [La API de internacionalización](#) [interpolación](#), [Literales de cadena](#)

[Clase Intl.DateTimeFormat](#), [Formateo de fechas y horas](#)[Fechas y horarios](#)

[Clase Intl.NumberFormat](#), [Formateo de números](#)-[Invocación condicional](#) de expresiones de invocación, [Invocación condicional](#), [Invocación](#) del método de invocación de funciones, [Expresiones de invocación](#), [Visión general](#) de la [invocación de métodos](#), [Invocación de funciones](#) [Función isFinite\(\)](#), [Aritmética en JavaScript](#) [Función isNaN\(\)](#), [Aritmética en JavaScript](#)

iteradores y generadores (véase también métodos de iteradores de matrices)

características avanzadas del generador valor de retorno de las funciones del generador, [El valor de retorno de una función del generador](#)

métodos `return()` y `throw()`, [Los métodos return\(\) y throw\(\) de un valor generador](#) de expresiones `yield`, [El valor de una expresión yield](#)

asíncrono, Iteradores [asíncronos-Implementación de Iteradores Asíncronos](#) cierre de iteradores, "Cerrar" un Iterador: El método de retorno generadores beneficios de, Una nota final sobre la creación de generadores, Generadores definición del término, Generadores ejemplos de, Generadores ejemplos generadores yield\* y recursivos, [yield\\* y recursivos Generadores](#) cómo funcionan los iteradores, [Cómo funcionan los iteradores](#)

implementación de objetos iterables, [Implementación de objetos iterables](#) Implementación de objetos iterables visión general de, [Iteradores y generadores](#)

## J

JavaScript  
beneficios de, [Introducción a JavaScript](#), [Resumen de introducción a los capítulos](#), [Un recorrido por JavaScript](#), [Un recorrido por JavaScript](#)  
histogramas de frecuencia de caracteres, Ejemplo:  
[Histogramas de frecuencia de caracteres-Resumen Hello World](#), Historia de [Hello World](#) de, [JavaScript en los navegadores web](#) Intérpretes de JavaScript, [Exploración de la estructura léxica de JavaScript](#), [Estructura léxica-Resumen de nombres, versiones y modos](#), [Introducción a JavaScript](#)

sintaxis y capacidades, [A Tour of JavaScript](#)-[A Tour of Documentación de referencia de](#)

[JavaScript](#), [Prefacio](#)

Biblioteca estándar de JavaScript

[API de la Consola](#), [La API de la Consola](#)-Salida formateada con la [Consola](#)

fechas y horas, [Dates and Times](#)-Formatting and Parsing Date [Strings](#) clases de error, [Error Classes](#)-[Error Classes](#)

[API de internacionalización](#), [La API de internacionalización](#)-Comparación de cadenas

Serialización y análisis sintáctico de JSON, [JSON Serialization and Parsing](#)[JSON Customizations](#) overview of, [The JavaScript Standard Library](#)

Coincidencia de patrones, [Coincidencia de patrones con expresiones regulares](#)[exec\(\)](#) conjuntos y mapas, [Conjuntos y mapas](#)-Temporizadores [WeakMap](#) y [WeakSet](#),

Temporizadores-Temporizadores

matrices tipificadas y datos binarios, [matrices tipificadas y datos binarios](#)Vista de datos y endogamia

[APIs de URLs](#), [APIs de URLs](#)-Funciones de URLs de Legacy

Jest, [Pruebas unitarias con el método](#)

[Jest join\(\)](#), [Conversiones de Array a](#)

[String](#)

Serialización y análisis sintáctico de JSON, [JSON Serialization and Parsing](#)-[JSON](#)

Personalizaciones

Función JSON.parse(), Serialización de objetos, Serialización de JSON y Analizar

Función JSON.stringify(), Serialización de objetos, El método toJSON(), Serialización y análisis de JSON

Extensión del lenguaje JSX, JSX: Markup Expressions in JavaScript-JSX: Markup Expressions in JavaScript jump statements, Statements, Jumps-try/catch/finally break statements, break continue statements, continue definición del término, Statements labeled statements, Labeled Statements overview of, Jumps return statements, return

## K

palabras clave

Palabra clave async, async y await-Detalles de la implementación Palabra clave await, async y await-Detalles de la implementación Sensibilidad a las mayúsculas y minúsculas, El texto de un programa JavaScript

Palabra clave class, Clases con la palabra clave class-Ejemplo: Una clase de números complejos palabra clave const, Declaraciones con la palabra clave let y const export, Módulos en ES6 palabra clave function, Definición de funciones palabra clave function\*, Generadores palabra clave import, Módulos en ES6

Palabra clave let, [Un recorrido por JavaScript](#), Declaraciones con let y const, const, let y var Palabra clave new, [Creación de objetos con new](#), Palabras reservadas de la invocación del constructor, Palabras reservadas, [Expresiones primarias](#)

Palabra clave this, [Un recorrido por JavaScript](#), Expresiones primarias, [Invocación de funciones](#) Palabra clave var, Declaraciones de variables con var, const, let y var Palabra clave yield\*, [generadores yield\\*](#) y recursivos

Copos de nieve Koch, [ejemplo de transformación](#)

## L

sentencias etiquetadas, [sentencias etiquetadas](#) propiedad lastIndex, [exec\(\)](#) método lastIndexOf(), [indexOf\(\)](#) y [lastIndexOf\(\)](#) operador menos que (<) visión general de, [Operadores de comparación](#) comparación de cadenas, [Trabajo con conversiones de tipo Strings](#), [Conversiones de operadores de casos especiales](#) operador menos que o igual a (≤) visión general de, [Operadores de comparación](#) comparación de cadenas, [Trabajo con conversiones de tipo Strings](#), [Conversiones de operadores de casos especiales](#)

La palabra clave let, [Un recorrido por JavaScript](#), Declaraciones con let y const, const, let y var lexical scoping, Estructura léxica de los cierres, Estructura léxica-Sensibilidad

a las mayúsculas y minúsculas, [El texto de un programa JavaScript](#) [comentarios, Comentarios identificadores](#), [El texto de un programa JavaScript - Identificadores y palabras reservadas](#) saltos de línea, [El texto de un programa JavaScript literales, Literales palabras reservadas, Palabras reservadas, Expresiones primarias punto y coma, Punto y coma opcional - Punto y coma opcional espacios](#), [El texto de un programa JavaScript Secuencias de escape del juego de caracteres Unicode, normalización de las secuencias de escape Unicode](#), visión general de la normalización Unicode, [Unicode saltos de línea, El texto de un programa JavaScript, Punto y coma opcional](#)[Estilos de línea de punto y coma opcional, Estilos de línea Terminadores de línea, El texto de un programa JavaScript herramientas de linting, Linting con literales ESLint numérico](#)

[literales de punto flotante, Literales de punto flotante, Punto flotante binario y errores de redondeo](#) [literales enteros, Literales enteros](#) números negativos, Separadores de números en, [Literales de punto flotante Expresiones regulares, Coincidencia de Patrones, Coincidencia de Patrones con Expresiones Regulares](#) string, [String Literals template literals, Template Literals, Template](#)

Tags little-endian architecture, DataView and Endianness  
load event, Ejecución de Programas JavaScript  
propiedad localStorage, localStorage y sessionStorage  
propiedad location, Location, Navigation, and History  
operadores lógicos, Un recorrido por JavaScript, Expresiones lógicas - Aserciones lookbehind lógicas (!), Especificación de la posición de coincidencia bucles do/while, bucles do/while for, for, Iteración de matrices  
Bucles for/await, Iteración asíncrona con for/await,  
Iteración asíncrona para bucles for/in, for/in,  
Enumeración de propiedades  
Bucles for/of, for/of/in, Iteración de matrices,  
Iteradores y generadores propósito de, Declaraciones  
while bucles, while lvalue, Operando y tipo de resultado

## M

magnetómetros, API para dispositivos móviles  
Conjunto de Mandelbrot, Ejemplo: El conjunto de Mandelbrot-Resumen y Sugerencias de lecturas adicionales  
Clase Map, for/of con Set y Map, La clase Map-La clase Map  
Objetos Map, Visión general y definiciones, La clase Map  
método map(), map()

marshaling, [JSON Serialization and Parsing](#) [match\(\)](#)

[method](#), [match\(\)](#) [matchAll\(\)](#) [method](#), [matchAll\(\)](#)

[matches\(\)](#) [method](#), [Selecting elements with CSS](#)

[selectors](#) [Math.pow](#) [function](#), [Arithmetic](#)

[Expressions](#)

operaciones matemáticas, [Aritmética en JavaScript](#)-[Aritmética en JavaScript](#)

Sitio web de MDN, [Prefacio de las APIs de](#)

[medios](#), [Memoización de las APIs de](#)

[medios](#), [Gestión de la memoria de](#)

[memoización](#), [Visión general y definiciones](#)

eventos de mensajes, [modelo de enhebrado del lado del cliente de JavaScript](#), [Eventos](#),

[Eventos enviados por el servidor](#), [objetos de los trabajadores: el objeto global en los trabajadores](#),

[Modelo de Ejecución del Trabajador: postMessage\(\)](#),

[MessagePorts](#) y [MessageChannels](#), [Mensajería cruzada con](#)

[postMessage\(\)](#), [fork\(\)-Hilos del Trabajador](#), [Canales de](#)

[Comunicación y MessagePorts](#), [Tipos Enumerados y Uniones](#)

[Discriminadas](#)

Canales de mensajes, [postMessage\(\)](#), [MessagePorts](#) y [MessageChannels](#)

Objetos [MessagePort](#), [postMessage\(\)](#), [MessagePorts](#), y

[MessageChannels](#), [Canales de Comunicación y mensajería](#)

[MessagePorts](#)

API de [WebSocket](#)

recepción de mensajes, [Recepción de mensajes desde un](#)

[WebSocket](#) envío de mensajes, [Envío de mensajes a través de un](#)

WebSocket hilos de trabajo y mensajería, Hilos de trabajo y mensajería

Mensajería de origen cruzado con postMessage() mensajería de origen cruzado, Mensajería de origen cruzado con postMessage(), Mensajería de origen cruzado con postMessage() modelo de ejecución, Modelo de ejecución de Worker importando código, Importando código en un Worker Ejemplo de conjunto de Mandelbrot, Ejemplo: El conjunto de MandelbrotResumen y sugerencias de lectura adicional de los módulos, Importación de código en un Worker visión general de, Hilos de trabajo y mensajería

postMessage(), MessagePorts y MessageChannels, postMessage(), MessagePorts y MessageChannels

Objetos de trabajo, Objetos de trabajo

Objeto WorkerGlobalScope, El objeto global en la metaprogramación de los trabajadores, Métodos de metaprogramación

Añadir métodos a clases existentes, Añadir métodos a clases existentes Métodos de arrays aplicación genérica de, Arrays visión general de, Métodos de arrays clase versus métodos de instancia, Creación de métodos estáticos, Un recorrido por JavaScript definición del término, Funciones, Invocación de métodos

encadenamiento de métodos, invocación de métodos

invocación de métodos, expresiones de invocación, métodos abreviados de invocación de métodos, sintaxis abreviada de Getters, Setters y otras formas de métodos, métodos abreviados métodos estáticos, métodos estáticos métodos de matrices tipificadas, métodos y propiedades de matrices tipificadas operador de sustracción con signo negativo (-), aritmética en JavaScript, expresiones aritméticas operador aritmético unario, operadores aritméticos unarios módulos de API para dispositivos móviles

automatización de la modularidad basada en cierres, automatización de la modularidad basada en cierres en ES6 importaciones dinámicas con import(), importaciones dinámicas con import() exportaciones, exportaciones ES6 import.meta.url, importaciones import.meta.url, importaciones ES6 importaciones y exportaciones con renombramiento, Importaciones y exportaciones con renombramiento

Módulos de JavaScript en la web, Módulos de JavaScript en la web - Visión general de los módulos en ES6 - Reexportaciones, Reexportaciones - Módulo fs (Node), Trabajo con archivos - Trabajo con directorios - Directivas de importación y exportación, Módulos

en Node, Módulos en Node-Módulos en la Web, Módulos de nodos

Exportaciones de nodos, Exportaciones de nodos

Importaciones de nodos, Importaciones de nodos

Módulos de estilo de nodo en la web, Módulos de estilo de nodo en la web

Visión general de los módulos, Propósito de los

módulos, Uso de los módulos en los

trabajadores, Importación de código en un

trabajador

con clases, objetos y cierres, Módulos con clases, objetos y

cierres-Automatización de la modularidad basada en cierres

operador modulo (%), Aritmética en JavaScript, Expresiones

aritméticas

operador de multiplicación (\*), Aritmética en JavaScript,

Expresiones y Operadores, Expresiones Aritméticas

programación multihilo, Node es asíncrono por defecto,

mutabilidad de los hilos de trabajo, visión general y

definiciones, introducción a los objetos

N

grupos de captura con nombre, Alternancia, agrupación

y referencias NaN (valor no numérico), Aritmética en

JavaScript navigator.mediaDevices.getUserMedia()

function, Media APIs navigator.vibrate() method, Mobile

Device APIs negative infinity value, Aritmética en

JavaScript negative zero, Aritmética en JavaScript nested

functions, Nested Functions network events, Network

Events networking, Networking-Protocol negotiation

Método fetch() abortando peticiones,

Abortando una petición cross-origin requests,

Cross-origin requests examples of, fetch() file

upload, File upload with fetch()

Códigos de estado HTTP, cabeceras de respuesta y errores de

red, Códigos de estado HTTP, cabeceras de respuesta y errores

de red opciones varias de la solicitud, Opciones varias de la

solicitud análisis de los cuerpos de la respuesta, Análisis de los

cuerpos de la respuesta configuración de las cabeceras de la

solicitud, Configuración de las cabeceras de la solicitud

configuración de los parámetros de la solicitud, Configuración de

los parámetros de la solicitud

especificar el método de solicitud y el cuerpo de la solicitud,

Especificar el método de solicitud y el cuerpo de la solicitud

pasos de, fetch()

cuerpos de respuesta de streaming, Visión general de los

cuerpos de respuesta de streaming, Eventos enviados por

el servidor de red, Eventos enviados por el servidor

API de WebSocket, WebSockets

API XMLHttpRequest (XHR), fetch() new keyword, Creación

de objetos con new, Invocación de constructores

new.target expression, Clases y constructores newline (\n),

Literales de cadena, Secuencias de escape en literales de

cadena newlines, Puntos y comas opcionales

utilizando para el formato del código, [El texto de un programa JavaScript](#)

Nodo

iteración asíncrona en, [El bucle for/await](#), [El nodo es asíncrono por defecto](#)

beneficios de, [Introducción a JavaScript](#), [JavaScript del lado del servidor con Node](#)

Tipo BigInt, [Enteros de precisión arbitraria con búferes](#)

[BigInt](#), [Devoluciones de llamada y eventos en búferes](#),

[Devoluciones de llamada y eventos en procesos hijo de](#)

[Node](#), [Trabajar con procesos hijo](#)-característica de

definición de [Fork\(\)](#), [JavaScript del lado del servidor con](#)

eventos de [Node](#) y [EventEmitter](#), [Eventos y EventEmitter](#)

manejo de archivos, [Trabajar con directorios](#) directorios,

[Trabajar con directorios](#)

metadatos de archivo, cadenas de

modo de archivo de [metadatos](#),

[escritura de archivos](#) operaciones de

archivo, resumen de [operaciones de](#)

[archivo](#), [trabajo con archivos](#)

[Rutas](#), [descriptores de archivos](#) y [FileHandles](#), [Rutas](#),

[descriptores de archivos](#) y [FileHandles](#) Lectura de

archivos, Lectura de archivos Escritura de archivos

[Clientes y servidores HTTP](#), [Clientes y servidores HTTP](#)-[Clientes y servidores HTTP](#)

instalación, Exploración de JavaScript, JavaScript del lado del servidor con Nodo

API Intl, Los módulos de la API de internacionalización en,

Módulos en Node-Node-Style Módulos en la Web

servidores y clientes de red no HTTP, Servidores y clientes de red no HTTP paralelismo con, Nodo es asíncrono por defecto detalles del proceso, Proceso, CPU y detalles del sistema operativo

fundamentos de la programación, fundamentos de la programación de nodos-el nodo

Argumentos de la línea de comandos del gestor de paquetes,

Argumentos de la línea de comandos y variables de entorno

salida de la consola, Variables de entorno de la salida de la

consola, Módulos del gestor de paquetes de Node, Ciclo de

vida del programa del gestor de paquetes de Node,

Documentación de referencia del ciclo de vida del programa,

Prefacio flujos, iteración asíncrona en modo de flujos,

Iteración asíncrona descripción general de flujos, Tubos,

Lectura de flujos con eventos, Tipos de flujos

escribir y manejar la contrapresión, Escribir en flujos y

Manejo de la contrapresión

hilos de trabajo, Worker Threads-Sharing Typed Arrays Between Canales de comunicación de hilos y MessagePorts, Canales de comunicación y MessagePorts

creación de trabajadores y paso de mensajes, Visión general de la creación de trabajadores y paso de mensajes, Hilos de trabajo

compartir matrices tipificadas entre hilos, Compartir matrices tipificadas entre hilos

transferir MessagePorts y matrices tipificadas, Transferir MessagePorts y matrices tipificadas

entorno de ejecución del trabajador, El entorno de ejecución del trabajador

NodeLists, Selección de elementos con selectores CSS

propiedades no heredadas, Introducción a los objetos

operador de desigualdad no estricto ( $\neq$ ) expresiones

relacionales, Operadores de igualdad y desigualdad

conversiones de tipo, Conversiones de operadores de

casos especiales normalización, Normalización Unicode

valor no numérico (NaN), Aritmética en JavaScript API de

notificaciones, Aplicaciones web progresivas y

trabajadores de servicios gestor de paquetes npm,

Gestión de paquetes con npm valores nulos, nulos e

indefinidos operador de coalescencia (??), Definido por

primera vez (??)

Tipo de número

Formato de coma flotante de 64 bits, Numbers

Números enteros de precisión arbitraria con BigInt, Números enteros de precisión arbitraria con BigInt

aritmética y matemáticas complejas, Aritmética en JavaScript-  
Aritmética en JavaScript

errores binarios de punto flotante y redondeo, [Binary Floating-Point and Rounding Errors](#) dates and times, [Dates and Times](#) floating-point literals, [Floating-Point Literals](#) literales enteros, separadores de literales enteros en literales numéricos, [literales en coma flotante](#) Función Number(), [Conversiones explícitas](#), [Conversiones explícitas](#) Función Number.isFinite(), [Aritmética en JavaScript](#) números, formato para la internacionalización, [Formato de los números](#)[Formato de los números](#) literales numéricos, [Números](#)

## O

objetos literales sintaxis extendida para, [Sintaxis literal de objeto extendida](#): descripción general de [Getters y Setters de propiedades](#), [Inicializadores de objetos y matrices](#) forma más simple de, [Literales de objetos](#) nombres de propiedades de objetos, [Lectura y escritura de elementos de matrices](#) programación orientada a objetos definición del término, [Descripción general y definiciones](#) ejemplo de, [Un recorrido por JavaScript](#)

Función Object.assign(), [Ampliación de objetos](#) Función Object.create(), [Object.create\(\)](#), [Atributos de las propiedades](#) Método Object.defineProperties(), [Atributos de las propiedades](#) Método Object.defineProperty(), [Atributos de la propiedad](#)

Método Object.entries(), para/de con objetos Función  
Object.getOwnPropertyNames(), Enumeración de propiedades  
Función Object.getOwnPropertySymbols(), Enumeración de propiedades

Método Object.keys, para/de con objetos

Función Object.keys(), Enumeración de propiedades Object.prototype, Prototipos,

Métodos de objetos

Arguments object, El objeto Arguments array-like objects,

Array-Like Objects-Array-Like Objects creating, Creación de objetos-Object.create() deleting properties,

Eliminación de propiedades enumerating properties,

Enumeración de propiedades

sintaxis literal de objeto extendido, Sintaxis literal de objeto extendidoConseguidores y fijadores de propiedades que extienden objetos, Extendiendo objetos

implementación de objetos iterables, Implementación de objetos iterablesImplementación de objetos iterablesintroducción a, Objetos

programación modular con, Módulos con clases, objetos y cierres

referencias de objetos mutables, Valores primitivos

inmutables y referencias de objetos mutables,

Introducción a los objetos nombrando propiedades dentro, Palabras reservadas expresiones de creación de objetos, Expresiones de creación de objetos extensibilidad

de objetos, [Extensibilidad de objetos](#) métodos de objetos,

[Métodos de objetos](#)-El método [toJSON\(\)](#)

[Visión general](#) de, [Un recorrido por JavaScript](#), [Visión general y definiciones](#)[Visión general y definiciones](#)

Consulta y establecimiento de propiedades, [Consulta y establecimiento de propiedades](#)[Errores de acceso a las propiedades](#) Serialización de objetos, [Serialización de objetos](#)  
[Prueba de propiedades](#), [Prueba de propiedades](#)

evento onmessage, [Recepción de mensajes desde un WebSocket](#), [Objetos Worker](#)-El objeto global en los Workers, [postMessage\(\)](#), [MessagePorts](#) y [MessageChannels](#), [Mensajería Cross-Origin](#) con operadores [postMessage\(\)](#) operadores aritméticos, [Un recorrido por JavaScript](#), [Aritmética en JavaScript](#)-[Aritmética en JavaScript](#), [Expresiones Aritméticas](#)[Operadores Bitwise](#)

[Operadores de asignación](#), [Expresiones de asignación](#)-[Asignación con operadores binarios de operación](#), [Operadores de comparación de número de operandos](#), [Operadores de comparación](#)

operadores de igualdad y desigualdad, [Operadores de igualdad y desigualdad](#) operadores de igualdad, [Un paseo por JavaScript](#)

formando expresiones con, [Un recorrido por JavaScript](#), [Expresiones y operadores](#) operadores lógicos, [Un recorrido por JavaScript](#), [Expresiones lógicas](#)[Operadores lógicos NOT \(!\)](#) operadores varios operador await, El [operador await](#) operador comma (,), El operador [comma \(,\) operador](#) condicional (?:), El [operador condicional \(?:\)](#) operador delete, El [operador delete](#)

operador first-defined (??), First-Defined (??)) operador typeof, El operador typeof operador void, El operador void número de operandos, Número de operandos tipo de operando y resultado, Tipo de operando y resultado asociatividad del operador, Asociatividad del operador precedencia del operador, Precedencia del operador efectos secundarios del operador orden de evaluación, Orden de evaluación visión general de, Visión general del operador operadores postfix, Punto y coma opcional operadores relacionales, Un recorrido por JavaScript, Expresiones relacionalesEl operador instanceof tabla de, Visión general de los operadores ternarios, Número de operandos puntos y comas optionales, Puntos y comas optionales desbordamiento, Aritmética en JavaScript propiedades propias, Introducción a los objetos, Herencia

## P

gestor de paquetes (Node), El gestor de paquetes de Node, Gestión de paquetes con paralelización npm, Node es asíncrono por defecto parametrización, Funciones función parseFloat(), Conversiones explícitas función parseInt(), Conversiones explícitas contraseñas, Rutas de almacenamiento, Rutas y polígonos-Concordancia de patrones de rutas y polígonos

Definición de expresiones regulares alternancia, agrupación y referencias, Alternancia, agrupación y referencias clases de caracteres, Banderas de clases de caracteres, Banderas de

caracteres literales, Aserciones de lookbehind de [caracteres literales](#), Especificación de capturas de grupos con nombre [de posición de coincidencia](#), Alternancia, agrupación y referencias de repetición no ávida, Especificaciones de patrones de repetición no ávida, [Definición de expresiones regulares](#) caracteres de repetición, [Repetición](#) que especifica la posición de coincidencia, [Especificación de la posición de coincidencia](#)

Clases de caracteres Unicode, Visión general de las [clases de caracteres](#), [Coincidencia de patrones con expresiones regulares](#) símbolos de coincidencia de patrones, Símbolos de coincidencia de patrones

Clase RegExp método exec(), [exec\(\)](#) propiedad lastIndex y reutilización de RegExp, [exec\(\)](#) visión general de, [La clase RegExp](#) propiedades [RegExp](#) método test(), [test\(\)](#) métodos de cadena para match(), [match\(\)](#) matchAll(), [matchAll\(\)](#) replace(), [replace\(\)](#) search(), [Métodos de cadena para la comparación de patrones](#)

split(), sintaxis de [split\(\)](#) para,

Patrones de [coincidencia](#), [Colores](#), [patrones y degradados](#)

API de solicitud de pago, criptografía y APIs  
relacionadas APIs de rendimiento, decapado de  
rendimiento, serialización y análisis de JSON  
píxeles, Coordenadas del documento y coordenadas de la  
ventana gráfica, Manipulación de píxeles operador de  
adición y asignación ( $+=$ ), Asignación con el operador de  
adición Operación, Aritmética en JavaScript, El operador +  
concatenación de cadenas, Literales de cadena, Trabajar con  
cadenas, El operador + conversiones de tipo, Conversiones de  
operadores de casos especiales operador aritmético unario,  
Operadores aritméticos unarios polígonos, Rutas y polígonos-  
Rutas y polígonos  
método pop(), pilas y colas con push(), pop(), shift() y unshift()  
Evento popstate, Categorías de eventos, Gestión del  
historial con pushState()-Cero positivo en red,  
Aritmética en JavaScript posesivos, Literales de  
cadena operadores postfix, Punto y coma opcional  
método postMessage(), postMessage(), MessagePorts y  
MessageChannels  
Prettier, Formato de JavaScript con  
expresiones primarias Prettier, Tipos  
primitivos de expresiones primarias  
Valores de verdad booleanos, Valores booleanos-Valores  
booleanos  
valores primitivos inmutables, valores primitivos inmutables y  
referencias a objetos mutables

Tipo de número, Resumen y definiciones de  
números-fechas y horas, Resumen y definiciones

Tipo de cadena, literales de plantillas con

etiquetas de texto función printprops(),

declaraciones de funciones campos

privados, campos públicos, privados y

estáticos procedimientos, funciones

programas

tratamiento de errores, errores de programa

Ejecución de JavaScript, Ejecución de programas de JavaScript-

Línea de tiempo de JavaScript del lado del cliente modelo de

enhebrado del lado del cliente, Modelo de enhebrado de

JavaScript del lado del cliente

Línea de tiempo del lado del cliente, Entrada y salida de la

Línea de tiempo JavaScript del lado del cliente, Entrada y

salida del programa

Aplicaciones web progresivas (PWA), aplicaciones web  
progresivas y servicio

Trabajadores

Cadenas de promesas, promesas, encadenamiento de promesas

Función Promise.all(), Promesas en paralelo

Promesas encadenadas, Promesas encadenadas-

Promesas encadenadas

manejo de errores con, Manejo de errores con Promesas, Más  
sobre

Promesas y errores-Los métodos catch y finally

hacer promesas, [hacer promesas-promesas en secuencia basadas](#) en otras promesas, [promesas basadas en otras promesas](#)

[basado](#) en valores síncronos, Promesas [basadas en valores síncronos](#) desde cero, Promesas [desde cero](#) visión general de, [Promesas](#) operaciones paralelas, [Promesas en Paralelo](#)

[Promesas en secuencia](#), [Promesas en secuencia-Promesas en secuencia](#)

Resolver promesas, [Resolver promesas-Más sobre promesas y errores](#) que regresan de las devoluciones de llamada de las promesas, [La terminología de los métodos catch y finally](#),

[Manejar errores con promesas](#) usando, [Usar promesas-Manejar errores con promesas](#) propiedades nombres de propiedades computados, [Nombres de propiedades computados](#) acceso condicional a propiedades, [Acceso condicional a propiedades](#) copiar de un objeto a otro, [Extender objetos](#)

definir las propiedades de su propia función, [Definir las propiedades de su propia función](#) definición del término, [Visión general y definiciones](#) borrar, [Borrar propiedades](#) enumerar propiedades, [Enumerar propiedades](#)

heredar, [herencia](#)

nomenclatura, [Símbolos](#), [Introducción a los objetos](#), [Símbolos como propiedad](#)  
[Nombres](#)



propiedades no heredadas, [Introducción a los objetos](#), [errores de acceso a la propiedad](#), [Errores de acceso a la propiedad](#), [expresiones de acceso a la propiedad](#), [Expresiones de acceso a la propiedad](#) atributos de la propiedad, [Introducción a los objetos](#), [Atributos de la propiedad](#) Descriptores de la propiedad de los atributos de la propiedad, [Conseguidores y fijadores de la propiedad](#), [Conseguidores y fijadores de la propiedad](#)

Consulta y configuración, [Consulta y configuración de propiedades](#)-Pruebas de [errores de acceso a propiedades](#), [Pruebas de propiedades de matrices tipificadas](#), [Métodos y propiedades de matrices tipificadas](#) método `propertyIsEnumerable()`, [Pruebas de propiedades](#) herencia prototípica, [Introducción a los objetos](#), Cadenas de prototipos de [herencia](#), [Prototipos](#) prototipos, [Prototipos](#), [Herencia](#), [La propiedad prototipo](#), Clases y prototipos, [El atributo prototipo](#) invariantes proxy, [Invariantes Proxy](#)

Objetos proxy, [Invariantes de los objetos proxy](#) Números pseudoaleatorios, [Criptografía y APIs relacionadas](#) Campos públicos, [Campos públicos](#), privados y estáticos

Push API, [Progressive Web Apps y Service Workers](#)

método push(), [Un recorrido por JavaScript, Pilas y colas con push\(\), pop\(\), shift\(\) y unshift\(\)](#)

## Q

método quadraticCurveTo(), [Curvas](#) método querySelector(),

[Selección de elementos con selectores CSS](#) método

querySelectorAll(), [Selección de elementos con selectores](#)

[CSS comillas](#)

comillas dobles ("), [literales de](#)

[cadena](#) comillas simples ('),

[literales de cadena](#)

## R

React, [JSX: Expresiones de marcado en](#)

[JavaScript rectángulos, Rectángulos](#) funciones

recursivas, [Invocación de funciones generadores](#)

recursivos, [yield\\*](#) y generadores recursivos

método reduce(), [reduce\(\) y reduceRight\(\)](#)

tipos de referencia, [valores primitivos inmutables y referencias a objetos mutables](#)

API de Reflect, [La API de Reflect](#)

Función Reflect.ownKeys(), [Enumeración de propiedades](#)

Clase RegExp

método exec(), exec() propiedad

lastIndex y reutilización de RegExp,

exec() visión general de, La clase

RegExp propiedades RegExp,

propiedades RegExp método test(),

test()

Tipo RegExp, Visión general y definiciones, Coincidencia de patrones, Coincidencia de patrones con expresiones regulares (véase también coincidencia de patrones) expresiones regulares, Coincidencia de patrones con expresiones regulares

(ver también concordancia de patrones) expresiones relacionales, Expresiones relacionales-El operador instanceof operadores relacionales, Un recorrido por el método replace() de JavaScript, Trabajar con cadenas función require(),

Importaciones de nodos palabras reservadas, Palabras reservadas, Expresiones primarias parámetros restantes,

Parámetros restantes y listas de argumentos de longitud variable

declaraciones de retorno, valores de retorno, Funciones

Método return(), "Cerrando" un Iterador: El método return, Los métodos return() y throw() de un método reverse() del generador, Un recorrido por JavaScript, errores de redondeo de reverse(), Errores de punto flotante binario y de redondeo

S

política del mismo origen, La política del mismo origen

gráficos vectoriales escalables (SVG), SVG: Scalable Vector Graphics-

Creación de imágenes SVG con JavaScript creación de imágenes

SVG con JavaScript, Creación de imágenes SVG con JavaScript visión general de, SVG: Scalable Vector Graphics scripting SVG, Scripting SVG

SVG en HTML, API ScreenOrientation de SVG en HTML,

desplazamientos de las API de dispositivos móviles,

desplazamiento de las coordenadas del documento y de las

coordenadas de la ventana gráfica, método scrollTo() de

desplazamiento, método search() de desplazamiento, métodos

de cadena para el almacenamiento del lado del cliente de

seguridad de coincidencia de patrones, almacenamiento que

compite con los objetivos de la programación web, el modelo

de seguridad web

Cross-Origin Resource Sharing (CORS), La política del mismo origen, Peticiones de origen cruzado cross-site scripting (XSS), APIs de criptografía cross-site scripting, Criptografía y APIs relacionadas defensa contra código malicioso, Lo que JavaScript no puede hacer

ataques de denegación de servicio, escritura en flujos y manejo

Contrapresión

Política del mismo origen, La política del

mismo origen características de la plataforma web

a investigar, Punto y coma de seguridad (;), Punto y

como opcional información sensible, API de sensores de almacenamiento, API de dispositivos móviles

serialización, Serialización de objetos, Serialización y análisis de JSON, Gestión del historial con pushState() eventos enviados por el servidor, Eventos enviados por el servidor JavaScript del lado del servidor, JavaScript en los navegadores web, JavaScript del lado del servidor con Node

ServiceWorkers, Progressive Web Apps y Service Workers

propiedad sessionStorage, localStorage y sessionStorage

Clase Set, for/of con Set y Map, La clase Set-La clase Set

Objetos Set, Visión general y

definiciones Constructor Set(),

La clase Set Función

setInterval(), Temporizadores

conjuntos y mapas definición de

conjuntos, La clase Set

Clase de mapa, La clase de mapa-La clase de

mapa visión general de, Conjuntos y mapas

Clase de conjuntos, La clase de conjuntos-La clase de  
conjuntos

Clases WeakMap y WeakSet, WeakMap y WeakSet

métodos setter, Getters y Setters de propiedades, Getters, Setters y  
otras formas de métodos función setTimeout(), Timers, Timers

método `setTransform()`, [Coordinate System Transforms shadow DOM](#), [Shadow DOM-Shadow DOM API shadows](#), [Sombras](#) operador de desplazamiento a la izquierda (`<<`), [Operadores Bitwise](#) desplazamiento a la derecha con operador de signo (`>>`), [Operadores Bitwise](#) desplazamiento a la derecha con operador de relleno de cero (`>>>`), [Operadores Bitwise](#) método `shift()`, [pilas y colas con push\(\), pop\(\), shift\(\) y unshift\(\)](#) métodos abreviados, [métodos abreviados](#), [Getters, Setters y otras formas de métodos](#)

Efectos secundarios, [Operador Efectos secundarios](#) comillas simples ('), [Literales de cadena](#) método `slice()`, [slice\(\)](#) método `some()`, `every()` y `some()` ordenación, [Comparación de cadenas](#) método `sort()`, [Invocación condicional](#), [sort\(\)](#) arrays dispersos, [Arrays](#), [Arrays dispersos](#) método `splice()`, [splice\(\)](#) método, [split\(\)](#)

operador de propagación (...), [Operador de propagación](#), [El operador de propagación](#), [El operador de propagación para llamadas a funciones](#), [Iteradores y generadores corchetes \(\[\]\)](#), [Un recorrido por JavaScript](#), [Trabajar con cadenas](#), [Inicializadores de objetos y matrices](#), [Consulta y establecimiento de propiedades](#), [Lectura y escritura de elementos de matrices](#), [Cadenas como matrices](#) biblioteca estándar (véase biblioteca

estándar de JavaScript) bloques de declaraciones, Declaraciones compuestas y vacías (véase también declaraciones) sentencias compuestas y vacías, sentencias condicionales compuestas y vacías, sentencias, condicionales-switch estructuras de control, Un recorrido por JavaScript Un recorrido por JavaScript, Declaraciones declaraciones de expresión, Declaraciones de expresión frente a expresiones, Un recorrido por JavaScript declaraciones if/else, Valores booleanos declaraciones de salto, Declaraciones, Saltos-try/catch/finally saltos de línea y, Puntos y coma opcionales-Lista de puntos y coma opcionales de, Resumen de las declaraciones de JavaScript bucles, Declaraciones, Bucles-for/in declaraciones varias declaraciones de depuración, depurador usar directiva estricta, "usar estricto" con declaraciones, Declaraciones misceláneas resumen de, Declaraciones separando con punto y coma, Puntos y comas opcionales sentencias throw, sentencias try/catch/finally, sentencias try/catch/finally-try/catch/finally yield, yield, El valor de una expresión yield campos estáticos, Campos públicos, privados y estáticos métodos estáticos storage, Almacenamiento-IndexedDB cookies, Cookies IndexedDB, IndexedDB localStorage y sessionStorage, localStorage y sessionStorage visión general

de, Storage security and privacy, Storage streams (Node),  
Streams-Paused mode asynchronous iteration in, Asynchronous Iteration overview of, Streams pipes, Pipes reading with events,  
Reading Streams with Events types of, Streams

escribir y manejar la contrapresión, Escribir en flujos y  
Manejo del operador de

igualdad estricta (==) valores

booleanos, Visión general de  
los valores booleanos,

Operadores de igualdad y  
desigualdad comparación de  
cadenas, Trabajar con cadenas

conversiones de tipo, Visión general y definiciones,  
Conversiones y  
Igualdad modo estricto aplicación por defecto de, Clases  
con la palabra clave class, Módulos en ES6, Módulos JavaScript en la Web operador delete y, El operador delete borrando  
propiedades, Borrando propiedades función eval(),  
Declaraciones de función eval() estricta, Declaraciones de función invocación de función, Invocación de función frente a  
modo no estricto, "use strict"- "use strict" optar por,  
Introducción a JavaScript

TypeError, Errores de acceso a propiedades,  
Extensibilidad de objetos variables no declaradas y,  
Declaraciones de variables con var con declaración y, con,

[Establecimiento de atributos de manejadores de eventos](#)

literales de cadena

secuencias de escape en, [Secuencias de escape en literales](#)

de cadena visión general de, [Literales de cadena](#)

Función String(), [Conversiones explícitas](#)

Función String.raw(), [Literales de plantilla](#)

[etiquetados cadenas](#)

Conversiones de matrices a cadenas,

[Conversiones de matrices a cadenas](#) caracteres y

puntos de código, Métodos de [texto para la](#)

comparación de patrones [match\(\)](#), [match\(\)](#)

[matchAll\(\)](#), [matchAll\(\)](#) [replace\(\)](#) [search\(\)](#), [Métodos](#)

[de cadenas para la comparación de patrones](#)

[split\(\)](#), [split\(\)](#)

Visión general de, [Literales de cadena de texto](#),

[Literales de cadena](#) cadenas como arrays, [Cadenas como](#)

[arrays](#) trabajando con el acceso a caracteres individuales,

[Trabajando con cadenas](#) API para, [Trabajando con cadenas](#)

comparando, [Trabajando con cadenas](#), [Comparando](#)

[cadenas](#) concatenación, [Trabajando con cadenas](#)

determinando la longitud, [Trabajando con cadenas](#)

inmutabilidad, [Trabajando con cadenas](#) algoritmo de

clonación estructurado, [Gestión del historial con pushState\(\)](#)

subarrays, [Subarrays con slice\(\)](#), [splice\(\)](#), [fill\(\)](#), y [copyWithin\(\)](#)

subclases jerarquías de clases y clases [abstractas](#), [Jerarquías de clases y clases abstractas](#)-resumen delegación frente a

herencia, [Delegación en lugar de herencia](#) visión general de,

[Subclases prototipos y](#), [Subclases y prototipos](#)

con cláusula extends, [Subclases con extends y super-](#)  
[Subclases con extends y subrutinas](#),

[Funciones](#) operador de sustracción (-),

[Aritmética en pares](#) sustitutos de

[JavaScript](#), [Texto](#)

SVG (véase gráficos vectoriales escalables

(SVG)) switch statements, [switch-switch](#)

[Symbol.asyncIterator](#), [Symbol.iterator](#) y [Symbol.asyncIterator](#)

[Symbol.hasInstance](#), [Symbol.hasInstance](#)

[Symbol.isConcatSpreadable](#), [Symbol.isConcatSpreadable](#)

[Symbol.iterator](#), [Símbolos conocidos](#)

Símbolo.species, [Símbolo.species-Símbolo.species](#)

[Symbol.toPrimitive](#), [Symbol.toPrimitive](#)

[Symbol.toStringTag](#), [Symbol.toStringTag](#)

Símbolo.unscopable, [Símbolo.unscopable](#)

Símbolos

definición de las extensiones del lenguaje, [Visión general y](#)

[definiciones](#) nombres de propiedades, [Símbolos](#), [Símbolos](#)

como nombres de propiedades símbolos conocidos,

[Símbolos conocidos](#) ejecución sincrónica de scripts, [Cuándo](#)

se ejecutan los scripts: estructuras de control de sintaxis  
asíncronas y diferidas, Un recorrido por JavaScript-Un recorrido por JavaScript declarando variables, Un recorrido por JavaScript

Comentarios en inglés, Un recorrido por JavaScript, Un recorrido por los operadores relacionales y de igualdad de JavaScript, Un recorrido por las expresiones de JavaScript que se forman con operadores, Un recorrido por la expresión inicializadora de JavaScript, Un recorrido por JavaScript

extendido para los literales de objetos, Sintaxis extendida de los literales de objetosFunciones Getters y Setters de propiedades, Un recorrido por la estructura léxica de JavaScript, Estructura léxica-  
Sensibilidad a mayúsculas y minúsculas, El texto de un programa JavaScript comentarios, Comentarios

identificadores, El texto de un programa JavaScript - Identificadores y palabras reservadas saltos de línea, El texto de un programa JavaScript literales, Literales palabras reservadas, Palabras reservadas, Expresiones primarias punto y coma, Punto y coma opcional - Punto y coma opcional espacios, El texto de un programa JavaScript

Conjunto de caracteres Unicode, Secuencias de escape  
Unicode-Unicode

Operadores lógicos [de normalización](#), [Recorrido por los métodos de JavaScript](#), [Recorrido por los objetos de JavaScript](#) con acceso condicional a las propiedades, [Recorrido por la declaración de JavaScript](#), [Recorrido por los métodos abreviados de JavaScript](#), [Declaraciones de métodos abreviados](#), [Recorrido por las variables de JavaScript](#), asignación de valores a, [Recorrido por JavaScript](#)

## T

pestañas, [El texto de un programa JavaScript](#) etiquetado literales de la plantilla, [Etiquetado literales de la plantilla](#), [Plantilla Etiquetas literales de la plantilla](#), [Plantilla Etiquetas operadores ternarios](#), [Número de Operandos](#) método `test()`, [test\(\) texto](#) dibujo en lienzo, [texto](#) secuencias de [escape](#) en literales de [cadena](#), [secuencias de escape en literales de cadena](#) concordancia de patrones, [concordancia de patrones](#) literales de cadena, [literales de cadena](#) que representan el tipo de cadena, literales de [plantilla de texto](#), [literales de plantilla que trabajan con cadenas](#), [trabajar con editores de texto de cadenas](#)

normalización, [normalización](#)

[Unicode](#) con Node, estilos de texto

[Hello World](#), estilos de texto

. then(), [Uso de promesas](#), [Encadenamiento de promesas](#), [Más sobre promesas y errores](#)

esta palabra clave, [Un recorrido por JavaScript](#), [Expresiones primarias](#), [Invocación de funciones](#)

threading, [Worker Threads and Messaging](#), [Progressive Web](#)

[Apps and Service Workers](#) (ver también Worker API) 3D

graphics, [Graphics in a < canvas>](#) sentencias throw, [throw](#),

[Clases de error](#) método throw(), [Los métodos return\(\) y throw\(\)](#)

[de un generador](#) husos horarios, [Formateo de fechas y horas](#)

temporizadores, [Temporizadores](#), [Timestamps](#), [Fechas y horas](#),

[Timestamps](#) método [toDateString\(\)](#), [Formateo y análisis de](#)

[cadenas de fechas](#) método [toExponential\(\)](#), [Conversiones](#)

[explícitas](#) método [toFixed\(\)](#), [Conversiones explícitas](#)

Método [toISOString\(\)](#), [Formateo y análisis de cadenas de](#)

[fechas](#), [Personalizaciones JSON](#) Método [toJSON\(\)](#), [El método](#)

[toJSON\(\)](#), [Personalizaciones JSON](#)

Método [toLocaleDateString\(\)](#), [Formateo y análisis de cadenas de](#)

[fechas](#), [Formateo de fechas y horas](#)

Método [toLocaleString\(\)](#), [El método toLocaleString\(\)](#), [Conversiones de](#)

[matrices a cadenas](#), [Formateo y análisis de cadenas de fechas](#)

método [toLocaleTimeString\(\)](#), [Formateo y análisis de cadenas de](#)

[fechas](#), [Formato de fechas y horas](#)

herramientas y extensiones, [herramientas y extensiones de JavaScript](#)

## [Tipos y uniones discriminadas](#)

agrupación de códigos,

### [agrupación de códigos](#)

Formato de JavaScript con Prettier, [Formato de JavaScript con Prettier](#)

Extensión del lenguaje JSX, [JSX: Markup Expressions in JavaScript](#) [JSX: Markup Expressions in JavaScript](#) linting with ESLint, [Linting with ESLint](#) overview of, [JavaScript Tools and Extensions](#) package management with npm, [Package Management with npm](#) transpilation with Babel,

### [Transpilation with Babel](#)

comprobación de tipos con Flow, [comprobación de tipos con Flow-Enumerated](#)

[Tipos y uniones discriminadas](#) tipos de matrices, [Tipos de matrices](#)

tipos de clases, [Tipos de clases](#) tipos

enumerados y uniones discriminadas, [Tipos](#)

[enumerados y uniones discriminadas](#) tipos de

funciones, [Tipos de funciones](#) instalación y

ejecución, [Instalación y ejecución de Flow](#) tipos de

objetos, [Tipos de objetos](#) otros tipos

parametrizados, [Otros tipos parametrizados](#) visión

general de, [Comprobación de tipos con Flow](#) tipos

de sólo lectura, [Tipos de sólo lectura](#) alias de tipos,

[Alias de tipos](#)

TypeScript versus Flow, Comprobación de tipos con tipos de unión de Flow, Tipos de unión usando anotaciones de tipo, Uso de anotaciones de tipo pruebas unitarias con Jest, Pruebas unitarias con el método toPrecision() de Jest, Conversiones explícitas Método toString(), Valores booleanos, Conversiones explícitas, Los métodos toString() y valueOf(), El operador +, Igualdad con la conversión de tipos, El método toString(), El método toTimeString(), Formateo y análisis de cadenas de fechas Método toTimeString(), Formateo y análisis de cadenas de fechas Método toUpperCase(), Trabajo con cadenas Método toUTCString(), Formateo y análisis de cadenas de fechas transformaciones, Transformación del sistema de coordenadas-Ejemplo de transformación método translate(), Transformación del sistema de coordenadas translucidez, transpilación de translucidez y composición, transpilación con valores de verdad de Babel, valores booleanos sentencias try/catch/finally, try/catch/finally-try/catch/finally comprobación de tipos, comprobación de tipos con tipos numerados de flujo y Uniones discriminadas tipos de matrices, Tipos de matrices tipos de clases, Tipos de clases

tipos [enumerados](#) y uniones [discriminadas](#), [Tipos enumerados](#) y uniones [discriminadas](#) tipos de función, [Tipos de función instalación](#) y ejecución de [Flow](#) tipos de objeto, [Tipos de objeto](#) otros tipos parametrizados, [Otros tipos parametrizados](#) descripción general de, [Comprobación de tipos con Flow](#) tipos de sólo lectura, [Tipos de sólo lectura](#) alias de tipos, [Alias de tipos](#) [TypeScript](#) versus [Flow](#), [Comprobación de tipos con Flow](#) tipos de unión, [Tipos de unión](#) usando anotaciones de tipo, [Usando anotaciones de tipo](#) conversiones de tipo igualdad y, [Conversiones e igualdad](#), [Igualdad con conversión de tipos](#) conversiones explícitas, [Conversiones explícitas](#) datos financieros y científicos, [Conversiones explícitas](#) conversiones implícitas, [Conversiones explícitas](#) algoritmos de conversión objeto a primitivo, [Conversiones objeto a primitivo](#), [Algoritmos de conversión](#) objeto a booleano, [Conversiones objeto a booleano](#) objeto a número, [Conversiones objeto a número](#) objeto a cadena, [Conversiones objeto a cadena](#) conversiones de operadores de casos [especiales](#), [Conversiones de operadores de casos especiales](#) métodos [toString\(\)](#) y [valueOf\(\)](#), Visión general de los métodos [toString\(\)](#) y [valueOf\(\)](#), [Conversiones de](#)

tipos creación de matrices tipadas, Creación de matrices tipadas

DataView y endianness, Métodos y propiedades de DataView y Endianness, Visión general de los métodos y propiedades de los arrays tipificados, Arrays tipificados y datos binarios frente a arrays normales, Arrays compartidos entre hilos, Compartir arrays tipificados entre hilos tipos de arrays tipificados, Tipos de arrays tipificados utilizando, Uso de arrays tipificados

operador typeof, El operador typeof tipos objeto global, El objeto global-El objeto global Tipo de número, Números-Fechas y Tiempos objetos (ver objetos)

Visión general de, Tipos, valores y variables-Visión general y definiciones primitiva, Visión general y definiciones RegExp, Coincidencia de patrones-Concordancia de patrones cadenas, Texto-Literales de plantilla etiquetados Símbolos, Símbolos-Símbolos

Conversiones de tipo, Conversiones de tipo - Algoritmos de conversión de objeto a primitivo

TypeScript, comprobación de tipos con Flow

U

Uint8Array, Tipos de matrices tipificadas, Cuerpos de respuesta en flujo, Bufferes operadores aritméticos unarios, Operadores aritméticos unarios Operador booleano NOT (!), Valores booleanos

Soporte de JavaScript para, Número de Operandos variables no declaradas, Declaraciones de Variables con valores indefinidos var, null y undefined

underflow, Aritmética en JavaScript underscore (\_),

Identificadores y palabras reservadas underscores, como separadores numéricos (\_), Literales en coma flotante unescape() function, Funciones URL heredadas unhandledrejection event, Errores de programa

Secuencias de escape del juego de caracteres Unicode,

Secuencias de escape Unicode, Secuencias de escape en literales de cadena Cadenas JavaScript, Normalización del texto, Visión general de la normalización Unicode, Coincidencia de patrones Unicode, Clases de caracteres Caracteres del espacio, El texto de un programa JavaScript pruebas unitarias, Pruebas unitarias con Jest

método unshift(), pilas y colas con push(), pop(), shift() y unshift()

APIs de URLs, APIs de URLs-Funciones de URLs de Legacy

uso de la directiva strict aplicación por defecto del modo strict,

Clases con la palabra clave class, Módulos en ES6, Módulos

JavaScript en la web operador delete y, El operador delete  
función eval(), Declaraciones de funciones eval() estrictas,  
Declaraciones de funciones  
  
invocación de funciones, Invocación de funciones  
optando por el modo estricto, Introducción a JavaScript  
modo estricto versus no estricto, "use strict" - "use strict"  
TypeError, Errores de acceso a propiedades,  
Extensibilidad de objetos variables no declaradas y,  
Declaraciones de variables con var con declaración y,  
con, Establecimiento de atributos de manejadores de  
eventos usar modo estricto, Introducción a JavaScript y  
variables globales, Declaraciones de variables con var  
borrar propiedades, Borrar propiedades  
  
Codificación UTF-16, Texto

## V

Método valueOf(), Los métodos toString() y valueOf(),  
La asignación de valores del método valueOf(), Un  
recorrido por los valores booleanos de JavaScript,  
Valores booleanos-Valores booleanos falsos y  
verdaderos, Valores booleanos

funciones como valores, Funciones como valores-  
Definiendo sus propias propiedades de función  
valores primitivos inmutables, Valores primitivos  
inmutables y referencias a objetos mutables null y

undefined, Visión general de null y undefined, Tipos, valores y variables - Visión general y definiciones tipos de, Un recorrido por la palabra clave var de JavaScript, Declaraciones de variables con var, const, let y varargs, Parámetros restantes y listas de argumentos de longitud variable

funciones de aridad variable, Parámetros restantes y listas de argumentos de longitud variable variables sensibilidad a mayúsculas y minúsculas, El texto de un programa JavaScript declaraciones y asignaciones con let y const, Declaraciones con let y const-Declaraciones y tipos declaraciones con var, Declaraciones de variables con var

Asignación de desestructuración, Asignación de desestructuraciónResumen de la asignación de desestructuración, Recorrido por las variables no declaradas en JavaScript, Declaraciones de variables con definición de término var, Declaración y asignación de variables izadas, Declaraciones de variables con denominación var, Palabras reservadas visión general de, Tipos, Valores y Variables-Visión General y Definiciones ámbito de, Ámbito de variables y constantes, Funciones anidadas

Funciones variádicas, Parámetros de reposo y Argumento de longitud variable

Listas

flujos de vídeo, Media API

viewport, [Coordenadas del documento y coordenadas de la ventana gráfica](#), [Tamaño de la ventana gráfica](#), [Tamaño del contenido y Posición de desplazamiento](#) operador void, [El operador void](#)

## W

Clase WeakMap, [WeakMap y WeakSet](#)

Clase WeakSet, [WeakMap y WeakSet](#)

[API de autenticación web](#), [Criptografía y APIs relacionadas](#) APIs asíncronas del entorno del navegador web, APIs de audio de [eventos](#), [API de audio-La API de WebAudio](#) ventajas de JavaScript, [JavaScript en los navegadores web](#)

[API de lienzo](#), [Gráficos en un <lienzo>](#) -Manipulación de píxeles [dimensiones y coordenadas del lienzo](#), Recorte de [dimensiones y coordenadas del lienzo](#), Recorte transformaciones del sistema de [coordenadas](#), [transformaciones del sistema](#) de [coordenadas](#)-operaciones de dibujo de [ejemplo](#), [operaciones de dibujo en el lienzo](#)

Atributos de los gráficos, [Atributos de los gráficos](#), [Guardar y restaurar el estado de los gráficos](#) resumen de, [Gráficos en un <lienzo>](#) rutas y polígonos, [Rutas y polígonos](#)

manipulación de píxeles, [Pixel Manipulation](#)

Geometría del documento y desplazamiento, [Geometría del documento y desplazamiento](#)-Tamaño de la ventana, tamaño del contenido y posición de desplazamiento

Píxeles CSS, [coordenadas del documento y coordenadas de la ventana gráfica](#)

determinar [el elemento en un punto](#), [determinar el elemento en un punto](#)

coordenadas del documento y coordenadas [de la ventana gráfica](#), [Coordenadas del documento y coordenadas de la ventana gráfica](#)

Consulta de la geometría de los elementos, [Consulta de la geometría de un elemento](#), [Desplazamiento](#)

tamaño de la ventana gráfica, tamaño del contenido y posición de desplazamiento, [Viewport](#)

Eventos de [tamaño](#), [tamaño del contenido](#) y [posición de desplazamiento](#), [Eventos-despacho de eventos personalizados](#) despachando eventos

personalizados, [Despacho de eventos personalizados](#) cancelación de eventos, [Categorías de eventos](#)

cancelación de eventos, [Invocación de manejadores de eventos](#), [Invocación de manejadores de eventos](#)

propagación de eventos, [Propagación de eventos](#)

visión general de, [Eventos](#) registro de manejadores de eventos, [Registro de manejadores de eventos APIs](#)

heredadas, [JavaScript en navegadores web](#) ubicación, navegación e historial, [Ubicación](#), [navegación e](#)

historial-manejo de historia con pushState() historial de navegación, Historial de navegación carga de nuevos documentos, Carga de nuevos documentos visión general de, Ubicación, navegación e historial Ejemplo de conjunto de Mandelbrot, Ejemplo: The Mandelbrot Set-Summary and Suggestions for Further Reading module-aware browsers, JavaScript Modules on the Web networking, Networking-Protocol negotiation fetch() method, fetch() overview of, Networking server-sent events, Server-Sent Events

API de WebSocket, visión general de WebSockets, JavaScript en los navegadores web gráficos vectoriales escalables (SVG), SVG: Scalable Vector Graphics- Creación de imágenes SVG con JavaScript creación de imágenes SVG con JavaScript, Creación de imágenes SVG con JavaScript visión general de, SVG: Scalable Vector Graphics scripting SVG, Scripting SVG SVG en HTML, SVG en HTML scripting CSS, Scripting CSS-CSS Animaciones y Eventos estilos CSS comunes, Scripting CSS estilos computados, Estilos computados

Animaciones y eventos CSS, [Animaciones y eventos](#)

[CSS](#) [clases CSS](#), [Clases CSS](#) estilos en línea,

Convenciones de nomenclatura de [los estilos en línea](#), [Hojas de estilo de scripting de los estilos en línea](#), [Hojas de estilo de scripting](#)

Documentos de scripting, [Documentos de scripting-Ejemplo:](#) Generación de un

Estructura y recorrido del documento [del índice](#),

[Estructura y recorrido del documento](#)

Generar dinámicamente tablas de contenido, [Ejemplo:](#)

[Generación de una tabla de contenidos](#) modificación del

contenido, [Contenido de los elementos como HTML](#)

modificación de la estructura, [Creación, inserción y](#)

[eliminación de nodos](#) visión general de, [Scripting de](#)

[documentos](#) consulta y configuración de atributos,

[Atributos](#) selección de elementos del documento, [Selección](#)

[de elementos del documento](#) almacenamiento, [Cookies de](#)

Storage-IndexedDB, [IndexedDB](#)

[localStorage](#) y [sessionStorage](#), [localStorage](#) y

[sessionStorage](#) resumen de, Seguridad y

privacidad [del almacenamiento](#),

[Almacenamiento](#)

componentes web, [Componentes web-Ejemplo: a < search-box>](#)

[Componente web](#)

[Elementos personalizados](#), [Elementos personalizados](#)

Nodos de DocumentFragment, Uso de  
plantillas HTML de Web Components, Visión  
general de las plantillas HTML, Web  
Components

ejemplo de caja de búsqueda, Ejemplo: un <  
search-box> Web Component shadow DOM,  
Shadow DOM using, Uso de las características de la  
plataforma web de Web Components para  
investigar APIs binarias, APIs binarias

APIs de criptografía y seguridad, Criptografía y APIs  
relacionadas

eventos, Eventos

HTML y CSS, APIs de medios HTML y  
CSS, APIs de medios para  
dispositivos móviles, APIs de  
dispositivos móviles

APIs de rendimiento, Rendimiento

Progressive Web Apps and ServiceWorkers,  
Seguridad de Progressive Web Apps y Service  
Workers, Seguridad

WebAssembly, WebAssembly

Características de los objetos de la ventana y del  
documento, Más documentos y  
Características

de las ventanas

## Fundamentos de la programación web

Modelo de Objetos del Documento (DOM), El Modelo de Objetos del Documento-El Modelo de Objetos del Documento

Ejecución de programas JavaScript, Ejecución de programas JavaScript-Línea de tiempo JavaScript del lado del cliente

objeto global en los navegadores web, El objeto global en los navegadores web

JavaScript en etiquetas HTML <script>, JavaScript en HTML

<script> Etiquetas-Carga de scripts bajo demanda

errores de programa, Errores de programa

entrada y salida, Scripts de entrada y salida del programa compartiendo espacios de nombres,

Scripts comparten un espacio de nombres

modelo de seguridad web, El modelo de seguridad web - Cross-site scripting

Hilos de trabajo y mensajería, Hilos de trabajo y mensajería  
Mensajería cruzada con postMessage()

herramientas para desarrolladores web,

Explorando JavaScript

Manifiesto web, aplicaciones web progresivas y trabajadores de servicios

API de Web Workers, modelo de hilos de JavaScript del lado del cliente, Worker

## Hilos

WebAssembly, WebAssembly

API de WebAudio, La API de WebAudio

WebRTC API, Media API

API de WebSocket

creación, conexión y desconexión de WebSockets,

Creación, conexión y desconexión de WebSockets descripción

general de, Negociación del protocolo WebSockets,

Negociación del protocolo recepción de mensajes, Recepción

de mensajes de un WebSocket envío de mensajes, Envío de

mensajes a través de un WebSocket bucles while, while con

sentencias, Sentencias varias

API para trabajadores

Mensajería de origen cruzado, Mensajería de

origen cruzado con errores postMessage(),

Errores en el modelo de ejecución de los

trabajadores, Modelo de ejecución de los

trabajadores importando código, Importando

código en un trabajador

Ejemplo del conjunto de Mandelbrot, Ejemplo: El conjunto

de Mandelbrot-Resumen y Sugerencias de Lectura

Adicional módulos, Importación de código en un trabajador

visión general de, Hilos de trabajo y mensajería

postMessage(), MessagePorts y MessageChannels,

postMessage(), MessagePorts y MessageChannels

Objetos de trabajo, [Objetos de trabajo](#)

Objeto WorkerGlobalScope, [El objeto global en los trabajadores](#)

atributo escribible, [Introducción a los objetos](#), [Atributos de propiedad](#)

## X

API XMLHttpRequest (XHR), [fetch\(\)](#)

XSS (cross-site scripting), [La política del mismo origen](#)

## Y

sentencias yield, [yield](#), [El valor de una expresión yield](#)

palabra clave yield\*, [yield\\*](#) y generadores recursivos

## Z

cero negativo cero, [Aritmética en](#)

[JavaScript](#) cero positivo, [Aritmética en](#)

[JavaScript](#) arrays basados en cero,

[Arrays](#)

## Sobre el autor

**David Flanagan** ha estado programando y escribiendo sobre JavaScript desde 1995. Vive con su mujer y sus hijos en el Noroeste del Pacífico, entre las ciudades de Seattle (Washington) y Vancouver (Columbia Británica). David es licenciado en ciencias e ingeniería informática por el Instituto Tecnológico de Massachusetts y trabaja como ingeniero de software en VMware.

## Colofón

El animal que aparece en la portada de *JavaScript: La guía definitiva*, séptima edición, es un rinoceronte de Java (*Rhinoceros sondaicus*). Las cinco especies de rinoceronte se distinguen por su gran tamaño, su gruesa piel acorazada, sus patas de tres dedos y su cuerno de uno o dos hocicos. El rinoceronte de Java se asemeja al rinoceronte indio y, al igual que esta especie, los machos tienen un solo cuerno. Sin embargo, los rinocerontes de Java son más pequeños y tienen una textura de piel única. Aunque en la actualidad sólo se encuentran en Indonesia, los rinocerontes de Java se extendían por todo el sureste de Asia. Viven en hábitats de selva tropical, donde se alimentan de abundantes hojas y hierbas y se esconden de las plagas de insectos, como las moscas chupasangre, metiéndose hasta el hocico en el agua o el barro.

El rinoceronte de Java mide una media de 1,80 metros de altura y puede llegar a los 3 metros de longitud, y los adultos pesan hasta 3.000 libras. Al igual que el rinoceronte indio, su piel gris parece estar separada en "placas", algunas de ellas con textura. La vida natural de un rinoceronte de Java se estima entre 45 y 50 años. Las hembras dan a luz cada 3-5 años, tras un periodo de gestación de 16 meses. Las crías pesan unos 45 kilos al nacer y permanecen con sus protectoras madres hasta 2 años.

Los rinocerontes suelen ser un animal algo abundante, ya que se adaptan a una serie de hábitats y en la edad adulta no tienen depredadores naturales. Sin embargo, los humanos los han cazado casi hasta su extinción. El folclore sostiene que el cuerno del rinoceronte posee poderes mágicos y afrodisíacos, y por ello

los rinocerontes son un objetivo principal para los cazadores furtivos. La población de rinocerontes de Java es la más precaria: a partir de 2020, los cerca de 70 animales que quedan de esta especie viven, bajo vigilancia, en el Parque Nacional de Ujung Kulon, en Java (Indonesia). Esta estrategia parece estar ayudando a garantizar la supervivencia de estos rinocerontes por el momento, ya que en un censo de 1967 sólo se contabilizaron 25.

Muchos de los animales que aparecen en la portada de O'Reilly están en peligro de extinción; todos ellos son importantes para el mundo.

La ilustración en color de la portada es obra de Karen Montgomery, basada en un grabado en blanco y negro de Dover Animals. Los tipos de letra de la portada son Gilroy y Guardian Sans. El tipo de letra del texto es Adobe Minion Pro; el del encabezamiento, Adobe Myriad Condensed; y el del código, Ubuntu Mono, de Dalton Maag.