

JAVASCRIPT ELOQUENTE

CUARTA EDICIÓN

Marijn Haverbeke

TRADUCIDO AL ESPAÑOL POR
Midudev



ELOQUENT JAVASCRIPT

4TH EDITION

Written by Marijn Haverbeke.

Licensed under a [Creative Commons attribution-noncommercial license](#). All code in this book may also be considered licensed under an [MIT license](#).

Illustrations by various artists: Cover by Péchane Sumi-e. Chapter illustrations by Madalina Tantareanu. Pixel art in Chapters 7 and 16 by Antonio Perdomo Pastor. Regular expression diagrams in Chapter 9 generated with [regexper.com](#) by Jeff Avallone. Village photograph in Chapter 11 by Fabrice Creuzot. Game concept for Chapter 16 by [Thomas Palef](#).

A paper version of Eloquent JavaScript, including a bonus chapter, is being brought out by [No Starch Press](#). They also sell a more polished EPUB version that includes the bonus chapter.

TABLE OF CONTENTS

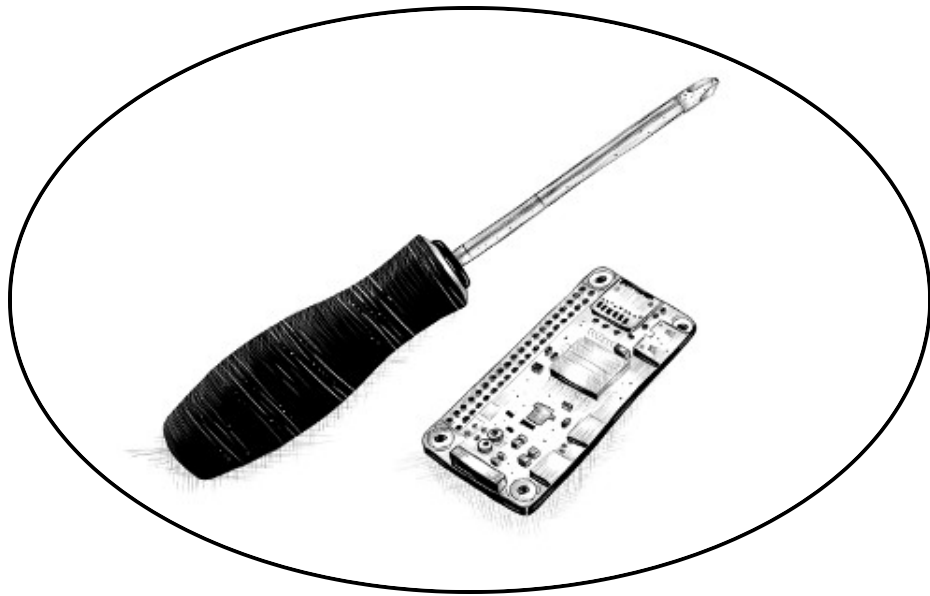
[Introduction](#)

2. [Values, Types, and Operators](#) (Part 1: Language)
 3. [Program Structure](#)
 4. [Functions](#)
 5. [Data Structures: Objects and Arrays](#)
 6. [Higher-order Functions](#)
 7. [The Secret Life of Objects](#)
 8. [Project: A Robot](#)
 9. [Bugs and Errors](#)
 10. [Regular Expressions](#)
 11. [Modules](#)
 12. [Asynchronous Programming](#)
 13. [Project: A Programming Language](#)
 14. [JavaScript and the Browser](#) (Part 2: Browser)
 15. [The Document Object Model](#)
 16. [Handling Events](#)
 17. [Project: A Platform Game](#)
 18. [Drawing on Canvas](#)
 19. [HTTP and Forms](#)
 20. [Project: A Pixel Art Editor](#)
 21. [Node.js](#) (Part 3: Node)
 22. [Project: Skill-Sharing Website](#)
- [Hints to the exercises](#)

INTRODUCCIÓN

“Creemos que estamos creando el sistema para nuestros propios propósitos. Creemos que lo estamos haciendo a nuestra propia imagen... Pero la computadora en realidad no es como nosotros. Es una proyección de una parte muy delgada de nosotros mismos: esa parte dedicada a la lógica, el orden, la regla y la claridad.”

—Ellen Ullman, *Cerca de la máquina: Tecnofilia y sus Descontentos*



Este es un libro sobre cómo instruir a computadoras. Las computadoras son tan comunes como los destornilladores hoy en día, pero son bastante más complejas, y hacer que hagan lo que quieres que hagan no siempre es fácil.

Si la tarea que tienes para tu computadora es común, bien entendida, como mostrarte tu correo electrónico o actuar como una calculadora, puedes abrir la aplicación correspondiente y ponerte a trabajar. Pero

para tareas únicas o abiertas, a menudo no hay una aplicación adecuada.

Ahí es donde entra en juego la programming. *Programar* es el acto de construir un *programa*—un conjunto de instrucciones precisas que le dicen a una computadora qué hacer. Debido a que las computadoras son bestias tontas y pedantes, programar es fundamentalmente tedioso y frustrante.

Afortunadamente, si puedes superar ese hecho—e incluso disfrutar del rigor de pensar en términos que las máquinas tontas pueden manejar—programar puede ser gratificante. Te permite hacer cosas en segundos que tardarían *una eternidad* a mano. Es una forma de hacer que tu herramienta informática haga cosas que antes no podía hacer. Además, se convierte en un maravilloso juego de resolución de acertijos y pensamiento abstracto.

La mayoría de la programación se realiza con lenguajes de programación. Un *lenguaje de programación* es un lenguaje artificialmente construido utilizado para instruir a las computadoras. Es interesante que la forma más efectiva que hemos encontrado para comunicarnos con una computadora se base tanto en la forma en que nos comunicamos entre nosotros. Al igual que los idiomas humanos, los lenguajes informáticos permiten combinar palabras y frases de nuevas formas, lo que permite expresar conceptos cada vez más nuevos.

En un momento dado, las interfaces basadas en lenguajes, como los mensajes de BASIC y DOS de los años 1980 y 1990, eran el principal método de interactuar con las computadoras. Para el uso informático rutinario, estas se han reemplazado en gran medida por interfaces

visuales, que son más fáciles de aprender pero ofrecen menos libertad. Pero si sabes dónde buscar, los lenguajes todavía están ahí. Uno de ellos, *JavaScript*, está integrado en cada navegador web moderno—y por lo tanto está disponible en casi todos los dispositivos.

Este libro intentará que te familiarices lo suficiente con este lenguaje para hacer cosas útiles y entretenidas con él.

SOBRE LA PROGRAMACIÓN

Además de explicar JavaScript, presentaré los principios básicos de la programación. La programación, resulta, es difícil. Las reglas fundamentales son simples y claras, pero los programas construidos sobre estas reglas tienden a volverse lo suficientemente complejos como para introducir sus propias reglas y complejidades. Estás construyendo tu propio laberinto, de alguna manera, y fácilmente puedes perderte en él.

Habrà momentos en los que leer este libro resulte terriblemente frustrante. Si eres nuevo en la programación, habrá mucho material nuevo que asimilar. Gran parte de este material luego se combinará de maneras que requieren que hagas conexiones adicionales.

Depende de ti hacer el esfuerzo necesario. Cuando te cueste seguir el libro, no saques conclusiones precipitadas sobre tus propias capacidades. Estás bien, simplemente necesitas seguir adelante. Tómate un descanso, vuelve a leer algo de material y asegúrate de leer y comprender los programas de ejemplo y los ejercicios. Aprender es un trabajo duro, pero todo lo que aprendas será tuyo y facilitará aún más el aprendizaje futuro.

“Cuando la acción se vuelve poco rentable, recopila información; cuando la información se vuelve poco rentable, duerme.”

Un programa es muchas cosas. Es un trozo de texto escrito por un programador, es la fuerza directiva que hace que la computadora haga lo que hace, es información en la memoria de la computadora, y al mismo tiempo controla las acciones realizadas en esta memoria. Las analogías que intentan comparar los programas con objetos familiares tienden a quedarse cortas. Una comparación vagamente adecuada es comparar un programa con una máquina: suelen estar implicadas muchas partes separadas y, para hacer que todo funcione, debemos considerar las formas en que estas partes se interconectan y contribuyen a la operación del conjunto.

Una computadora es una máquina física que actúa como anfitriona de estas máquinas inmateriales. Las computadoras mismas solo pueden hacer cosas increíblemente sencillas. La razón por la que son tan útiles es que hacen estas cosas a una velocidad increíblemente alta. Un programa puede combinar ingeniosamente un número enorme de estas acciones simples para hacer cosas muy complicadas.

Un programa es una construcción del pensamiento. Es gratuito de construir, es liviano y crece fácilmente bajo nuestras manos al teclear. Pero a medida que un programa crece, también lo hace su complejidad. La habilidad de programar es la habilidad de construir programas que no te confundan a ti mismo. Los mejores programas son aquellos que logran hacer algo interesante mientras siguen siendo fáciles de entender.

Algunos programadores creen que esta complejidad se gestiona mejor utilizando solo un conjunto pequeño de técnicas bien

comprendidas en sus programas. Han compuesto reglas estrictas (“mejores prácticas”) que prescriben la forma que deberían tener los programas y se mantienen cuidadosamente dentro de su pequeña zona segura.

Esto no solo es aburrido, sino que es ineficaz. A menudo, nuevos problemas requieren soluciones nuevas. El campo de la programación es joven y aún se está desarrollando rápidamente, y es lo suficientemente variado como para tener espacio para enfoques radicalmente diferentes. Hay muchos errores terribles que cometer en el diseño de programas, y deberías ir y cometerlos al menos una vez para entenderlos. Una noción de cómo es un buen programa se desarrolla con la práctica, no se aprende de una lista de reglas.

POR QUÉ IMPORTA EL LENGUAJE

Al principio, en los inicios de la informática, no existían los lenguajes de programación. Los programas lucían algo así:

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

Este es un programa para sumar los números del 1 al 10 y mostrar el resultado: $1 + 2 + \dots + 10 = 55$. Podría ejecutarse en una máquina hipotética simple. Para programar los primeros ordenadores, era necesario configurar grandes conjuntos de

interruptores en la posición correcta o perforar agujeros en tiras de cartón y alimentarlos al ordenador. Puedes imaginar lo tedioso y propenso a errores que era este procedimiento. Incluso escribir programas simples requería mucha astucia y disciplina. Los complejos eran casi inconcebibles.

Por supuesto, introducir manualmente estos patrones arcanos de bits (los unos y ceros) hacía que el programador se sintiera como un mago poderoso. Y eso debe valer algo en términos de satisfacción laboral.

Cada línea del programa anterior contiene una única instrucción. Podría escribirse en inglés de la siguiente manera:

1. Almacena el número 0 en la ubicación de memoria 0.
2. Almacena el número 1 en la ubicación de memoria 1.
3. Almacena el valor de la ubicación de memoria 1 en la ubicación de memoria 2.
4. Resta el número 11 al valor en la ubicación de memoria 2.
5. Si el valor en la ubicación de memoria 2 es el número 0, continúa con la instrucción 9.
6. Suma el valor de la ubicación de memoria 1 a la ubicación de memoria 0.
7. Añade el número 1 al valor de la ubicación de memoria 1.
8. Continúa con la instrucción 3.
9. Muestra el valor de la ubicación de memoria 0.

Aunque eso ya es más legible que la sopa de bits, sigue siendo bastante oscuro. Usar nombres en lugar de números para las instrucciones y las ubicaciones de memoria ayuda:

```

    Establecer "total" en 0.
    Establecer "count" en 1.
[bucle]
    Establecer "compare" en "count".
    Restar 11 de "compare".
    Si "compare" es cero, continuar en [fin].
    Sumar "count" a "total".
    Añadir 1 a "count".
    Continuar en [bucle].
[fin]
    Mostrar "total".

```

¿Puedes ver cómo funciona el programa en este punto? Las dos primeras líneas asignan los valores iniciales a dos ubicaciones de memoria: `total` se utilizará para construir el resultado de la computación, y `count` llevará la cuenta del número que estamos observando en ese momento. Las líneas que utilizan `compare` probablemente sean las más confusas. El programa quiere ver si `count` es igual a 11 para decidir si puede dejar de ejecutarse. Debido a que nuestra máquina hipotética es bastante primitiva, solo puede comprobar si un número es cero y tomar una decisión en función de ese valor. Por lo tanto, utiliza la ubicación de memoria etiquetada como `compare` para calcular el valor de `count - 11` y tomar una decisión basada en ese valor. Las siguientes dos líneas suman el valor de `count` al resultado e incrementan `count` en 1 cada vez que el programa decide que `count` aún no es 11. Aquí está el mismo programa en JavaScript:

```

let total = 0, count = 1;
while (count <= 10) {
    total += count;
    count += 1;
}
console.log(total);
// → 55

```

Esta versión nos proporciona algunas mejoras. Lo más importante es que ya no es necesario especificar la forma en que queremos que el programa salte hacia adelante y hacia atrás; la construcción `while` se encarga de eso. Continúa ejecutando el bloque (entre llaves) debajo de él siempre y cuando se cumpla la condición que se le ha dado. Esa condición es `count <= 10`, lo que significa “el recuento es menor o igual a 10”. Ya no tenemos que crear un valor temporal y compararlo con cero, lo cual era simplemente un detalle no interesante. Parte del poder de los lenguajes de programación es que pueden encargarse de los detalles no interesantes por nosotros.

Al final del programa, después de que la construcción `while` haya terminado, se utiliza la operación `console.log` para escribir el resultado.

Finalmente, así es como podría verse el programa si tuviéramos a nuestra disposición las operaciones convenientes `rango` y `suma`, que respectivamente crean una colección de números dentro de un rango y calculan la suma de una colección de números:

```
console.log(suma(rango(1, 10)));  
// → 55
```

La moraleja de esta historia es que el mismo programa puede expresarse de formas largas y cortas, ilegibles y legibles. La primera versión del programa era extremadamente oscura, mientras que esta última es casi en inglés: registra la suma del rango de números del 1 al 10. (Veremos en [capítulos posteriores](#) cómo definir operaciones como `suma` y `rango`.)

Un buen lenguaje de programación ayuda al programador al permitirle hablar sobre las acciones que la computadora debe realizar a un nivel más alto. Ayuda a omitir detalles, proporciona bloques de construcción convenientes (como `while` y `console.log`), te permite definir tus propios bloques de construcción (como `suma` y `rango`), y hace que esos bloques sean fáciles de componer.

¿QUÉ ES JAVASCRIPT?

JavaScript fue introducido en 1995 como una forma de agregar programas a páginas web en el navegador Netscape Navigator. Desde entonces, el lenguaje ha sido adoptado por todos los demás navegadores web gráficos principales. Ha hecho posibles aplicaciones web modernas, es decir, aplicaciones con las que puedes interactuar directamente sin tener que recargar la página para cada acción. JavaScript también se utiliza en sitios web más tradicionales para proporcionar distintas formas de interactividad e ingenio.

Es importante tener en cuenta que JavaScript casi no tiene nada que ver con el lenguaje de programación llamado Java. El nombre similar fue inspirado por consideraciones de marketing en lugar de un buen juicio. Cuando se estaba introduciendo JavaScript, el lenguaje Java se estaba comercializando mucho y ganaba popularidad. Alguien pensó que era una buena idea intentar aprovechar este éxito. Ahora estamos atrapados con el nombre.

Después de su adopción fuera de Netscape, se escribió un documento estándar para describir la forma en que debería funcionar el lenguaje JavaScript para que las diversas piezas de software que afirmaban

soportar JavaScript pudieran asegurarse de que realmente proporcionaban el mismo lenguaje. Esto se llama el estándar ECMAScript, según la organización Ecma International que llevó a cabo la estandarización. En la práctica, los términos ECMAScript y JavaScript se pueden usar indistintamente, son dos nombres para el mismo lenguaje.

Hay quienes dirán cosas *terribles* sobre JavaScript. Muchas de esas cosas son ciertas. Cuando me pidieron que escribiera algo en JavaScript por primera vez, rápidamente llegué a detestarlo. Aceptaba casi cualquier cosa que escribía pero lo interpretaba de una manera completamente diferente a lo que yo quería decir. Esto tenía mucho que ver con el hecho de que no tenía ni idea de lo que estaba haciendo, por supuesto, pero hay un problema real aquí: JavaScript es ridículamente liberal en lo que permite. La idea detrás de este diseño era que haría la programación en JavaScript más fácil para principiantes. En realidad, esto hace que encontrar problemas en tus programas sea más difícil porque el sistema no te los señalará.

Esta flexibilidad también tiene sus ventajas. Deja espacio para técnicas imposibles en lenguajes más rígidos y permite un estilo de programación agradable e informal. Después de aprender el lenguaje adecuadamente y trabajar con él durante un tiempo, he llegado a realmente *gustarme* JavaScript.

Ha habido varias versiones de JavaScript. La versión ECMAScript 3 fue la versión ampliamente soportada durante el ascenso al dominio de JavaScript, aproximadamente entre 2000 y 2010. Durante este tiempo, se estaba trabajando en una versión ambiciosa 4, la cual planeaba una serie de mejoras y extensiones radicales al lenguaje.

Cambiar un lenguaje vivo y ampliamente utilizado de esa manera resultó ser políticamente difícil, y el trabajo en la versión 4 fue abandonado en 2008. Una versión mucho menos ambiciosa 5, que solo realizaba algunas mejoras no controversiales, salió en 2009. En 2015, salió la versión 6, una actualización importante que incluía algunas de las ideas previstas para la versión 4. Desde entonces, hemos tenido nuevas actualizaciones pequeñas cada año.

El hecho de que JavaScript esté evolucionando significa que los navegadores tienen que mantenerse constantemente al día. Si estás usando un navegador más antiguo, es posible que no admita todas las funciones. Los diseñadores del lenguaje se aseguran de no realizar cambios que puedan romper programas existentes, por lo que los nuevos navegadores aún pueden ejecutar programas antiguos. En este libro, estoy utilizando la versión 2023 de JavaScript.

Los navegadores web no son las únicas plataformas en las que se utiliza JavaScript. Algunas bases de datos, como MongoDB y CouchDB, utilizan JavaScript como su lenguaje de secuencias de comandos y consulta. Varias plataformas para programación de escritorio y servidores, especialmente el proyecto Node.js (el tema del [Capítulo 20](#)), proporcionan un entorno para programar en JavaScript fuera del navegador.

CÓDIGO Y QUÉ HACER CON ÉL

El *código* es el texto que constituye los programas. La mayoría de los capítulos en este libro contienen bastante código. Creo que leer código y escribir código son partes indispensables de aprender a

programar. Intenta no solo echar un vistazo a los ejemplos, léelos atentamente y entiéndelos. Esto puede ser lento y confuso al principio, pero te prometo que pronto le tomarás la mano. Lo mismo ocurre con los ejercicios. No des por sentado que los entiendes hasta que hayas escrito realmente una solución que funcione.

Te recomiendo que pruebes tus soluciones a los ejercicios en un intérprete de JavaScript real. De esta manera, obtendrás comentarios inmediatos sobre si lo que estás haciendo funciona, y, espero, te tentará a experimentar y a ir más allá de los ejercicios.

La forma más sencilla de ejecutar el código de ejemplo en el libro —y de experimentar con él— es buscarlo en la versión en línea del libro en <https://eloquentjavascript.net>. Allí, puedes hacer clic en cualquier ejemplo de código para editarlo y ejecutarlo, y ver la salida que produce. Para trabajar en los ejercicios, ve a <https://eloquentjavascript.net/code>, que proporciona el código inicial para cada ejercicio de programación y te permite ver las soluciones.

Ejecutar los programas definidos en este libro fuera del sitio web del libro requiere cierto cuidado. Muchos ejemplos son independientes y deberían funcionar en cualquier entorno de JavaScript. Pero el código en los capítulos posteriores a menudo está escrito para un entorno específico (navegador o Node.js) y solo puede ejecutarse allí. Además, muchos capítulos definen programas más grandes, y las piezas de código que aparecen en ellos dependen unas de otras o de archivos externos. El [sandbox](#) en el sitio web proporciona enlaces a archivos ZIP que contienen todos los scripts y archivos de datos necesarios para ejecutar el código de un capítulo dado.

VISIÓN GENERAL DE ESTE LIBRO

Este libro consta aproximadamente de tres partes. Los primeros 12 capítulos tratan sobre el lenguaje JavaScript. Los siguientes siete capítulos son acerca de los navegadores web y la forma en que se utiliza JavaScript para programarlos. Por último, dos capítulos están dedicados a Node.js, otro entorno para programar en JavaScript. Hay cinco *capítulos de proyectos* en el libro que describen programas de ejemplo más grandes para darte una idea de la programación real.

La parte del lenguaje del libro comienza con cuatro capítulos que introducen la estructura básica del lenguaje JavaScript. Discuten las [estructuras de control](#) (como la palabra `while` que viste en esta introducción), las [funciones](#) (escribir tus propios bloques de construcción) y las [estructuras de datos](#). Después de estos, serás capaz de escribir programas básicos. Luego, los Capítulos [5](#) y [6](#) introducen técnicas para usar funciones y objetos para escribir código más *abstracto* y mantener la complejidad bajo control. Después de un [primer capítulo del proyecto](#) que construye un robot de entrega rudimentario, la parte del lenguaje del libro continúa con capítulos sobre [manejo de errores y corrección de errores](#), [expresiones regulares](#) (una herramienta importante para trabajar con texto), [modularidad](#) (otra defensa contra la complejidad) y [programación asíncrona](#) (tratando con eventos que toman tiempo). El [segundo capítulo del proyecto](#), donde implementamos un lenguaje de programación, concluye la primera parte del libro.

La segunda parte del libro, de los capítulos [13](#) a [19](#), describe las herramientas a las que tiene acceso JavaScript en un navegador. Aprenderás a mostrar cosas en la pantalla (Capítulos [14](#) y [17](#)), responder a la entrada del usuario ([Capítulo 15](#)) y comunicarte a través de la red ([Capítulo 18](#)). Nuevamente hay dos capítulos de proyecto en esta parte, construyendo un [juego de plataforma](#) y un [programa de pintura de píxeles](#).

El [Capítulo 20](#) describe Node.js, y el [Capítulo 21](#) construye un pequeño sitio web utilizando esa herramienta.

CONVENCIONES TIPOGRÁFICAS

En este libro, el texto escrito en una fuente monoespaciada representará elementos de programas. A veces estos son fragmentos autosuficientes, y a veces simplemente se refieren a partes de un programa cercano. Los programas (de los cuales ya has visto algunos) se escriben de la siguiente manera:

```
function factorial(n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return factorial(n - 1) * n;  
    }  
}
```

A veces, para mostrar la salida que produce un programa, la salida esperada se escribe después, con dos barras inclinadas y una flecha al frente.

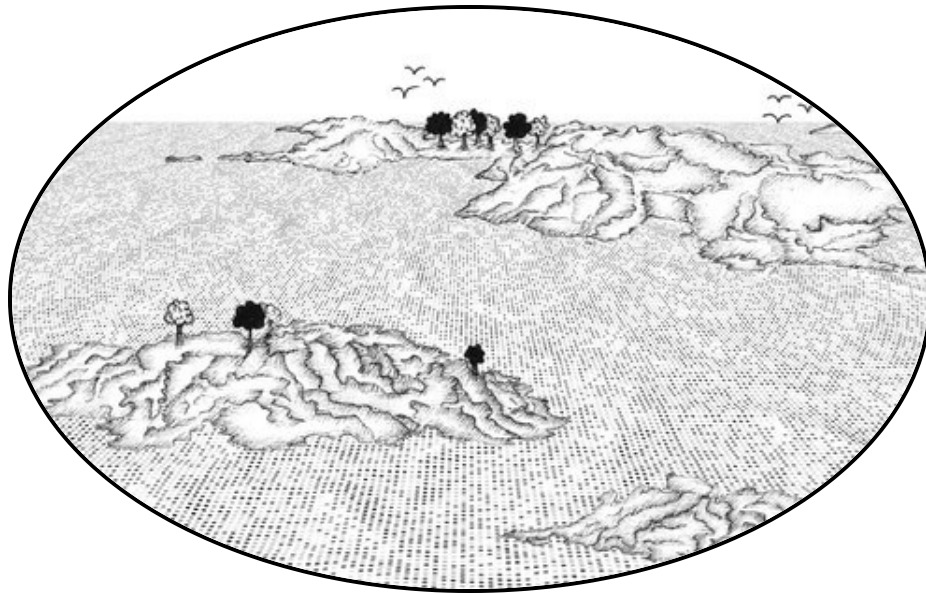
```
console.log(factorial(8));  
// → 40320
```

¡Buena suerte!

VALORES, TIPOS Y OPERADORES

“Debajo de la superficie de la máquina, el programa se mueve. Sin esfuerzo, se expande y contrae. En gran armonía, los electrones se dispersan y se reagrupan. Las formas en el monitor no son más que ondas en el agua. La esencia permanece invisible debajo.”

—Master Yuan-Ma, *El Libro de la Programación*



En el mundo de la computadora, solo existe data. Puedes leer data, modificar data, crear nueva data, pero aquello que no es data no puede ser mencionado. Toda esta data se almacena como largas secuencias de bits y, por lo tanto, es fundamentalmente similar.

Los bits son cualquier tipo de cosas de dos valores, generalmente descritos como ceros y unos. Dentro de la computadora, toman

formas como una carga eléctrica alta o baja, una señal fuerte o débil, o un punto brillante o opaco en la superficie de un CD. Cualquier pieza de información discreta puede reducirse a una secuencia de ceros y unos y por lo tanto representarse en bits.

Por ejemplo, podemos expresar el número 13 en bits. Esto funciona de la misma manera que un número decimal, pero en lugar de diez dígitos diferentes, tenemos solo 2, y el peso de cada uno aumenta por un factor de 2 de derecha a izquierda. Aquí están los bits que componen el número 13, con los pesos de los dígitos mostrados debajo de ellos:

0	0	0	0	1	1	0	1
128	64	32	16	8	4	2	1

Ese es el número binario 00001101. Sus dígitos no nulos representan 8, 4 y 1, y suman 13.

VALORES

Imagina una mar de bits—un océano de ellos. Una computadora moderna típica tiene más de 100 mil millones de bits en su almacenamiento de datos volátil (memoria de trabajo). El almacenamiento no volátil (el disco duro o equivalente) tiende a tener aún unos cuantos órdenes de magnitud más.

Para poder trabajar con tales cantidades de bits sin perderse, los separamos en trozos que representan piezas de información. En un entorno de JavaScript, esos trozos se llaman *valores*. Aunque todos los valores están hechos de bits, desempeñan roles diferentes. Cada valor tiene un tipo que determina su función. Algunos valores son

números, otros son fragmentos de texto, otros son funciones, y así sucesivamente.

Para crear un valor, simplemente debes invocar su nombre. Esto es conveniente. No tienes que recolectar material de construcción para tus valores ni pagar por ellos. Solo solicitas uno, y ¡zas!, lo tienes. Por supuesto, los valores no se crean realmente de la nada. Cada uno tiene que almacenarse en algún lugar, y si deseas usar gigantescas cantidades de ellos al mismo tiempo, podrías quedarte sin memoria de computadora. Afortunadamente, este es un problema solo si los necesitas todos simultáneamente. Tan pronto como dejes de usar un valor, se disipará, dejando atrás sus bits para ser reciclados como material de construcción para la próxima generación de valores. El resto de este capítulo presenta los elementos atómicos de los programas de JavaScript, es decir, los tipos de valores simples y los operadores que pueden actuar sobre dichos valores.

NÚMEROS

Los valores del tipo *number* son, como era de esperar, valores numéricos. En un programa de JavaScript, se escriben de la siguiente manera:

13

Usar esto en un programa hará que el patrón de bits para el número 13 exista en la memoria del ordenador.

JavaScript utiliza un número fijo de bits, 64 de ellos, para almacenar un único valor numérico. Solo hay tantos patrones que puedes hacer con 64 bits, lo que limita la cantidad de números diferentes que se

pueden representar. Con N dígitos decimales, puedes representar 10^N números. De manera similar, dada una cifra de 64 dígitos binarios, puedes representar 2^{64} números diferentes, que son alrededor de 18 mil trillones (un 18 seguido de 18 ceros). Eso es mucho.

La memoria de la computadora solía ser mucho más pequeña, y la gente solía utilizar grupos de 8 o 16 bits para representar sus números. Era fácil tener un *desbordamiento* accidental con números tan pequeños, terminando con un número que no encajaba en la cantidad dada de bits. Hoy en día, incluso las computadoras que caben en tu bolsillo tienen mucha memoria, por lo que puedes utilizar trozos de 64 bits y solo necesitas preocuparte por el desbordamiento cuando lidias con números realmente astronómicos.

Sin embargo, no todos los números enteros menores que 18 mil trillones encajan en un número de JavaScript. Esos bits también almacenan números negativos, por lo que un bit indica el signo del número. Un problema más grande es representar números no enteros. Para hacer esto, algunos de los bits se utilizan para almacenar la posición del punto decimal. El número entero máximo real que se puede almacenar está más en el rango de 9 cuatrillones (15 ceros), que sigue siendo increíblemente grande.

Los números fraccionarios se escriben usando un punto:

9.81

Para números muy grandes o muy pequeños, también puedes usar notación científica agregando una e (de *exponente*), seguida del exponente del número:

2.998e8

Eso es $2.998 \times 10^8 = 299,800,000$.

Los cálculos con números enteros (también llamados *enteros*) que son más pequeños que los mencionados 9 cuatrillones siempre serán precisos. Desafortunadamente, los cálculos con números fraccionarios generalmente no lo son. Así como π (pi) no puede expresarse con precisión mediante un número finito de dígitos decimales, muchos números pierden algo de precisión cuando solo están disponibles 64 bits para almacenarlos. Es una lástima, pero solo causa problemas prácticos en situaciones específicas. Lo importante es ser consciente de esto y tratar los números digitales fraccionarios como aproximaciones, no como valores precisos.

ARITMÉTICA

Lo principal que se puede hacer con los números es la aritmética. Operaciones aritméticas como la suma o la multiplicación toman dos valores numéricos y producen un nuevo número a partir de ellos. Así es como se ven en JavaScript:

```
100 + 4 * 11
```

Los símbolos $+$ y $*$ se llaman *operadores*. El primero representa la suma y el segundo representa la multiplicación. Colocar un operador entre dos valores aplicará ese operador a esos valores y producirá un nuevo valor.

¿Significa este ejemplo “Sumar 4 y 100, y luego multiplicar el resultado por 11”, o se realiza primero la multiplicación antes de la

suma? Como habrás adivinado, la multiplicación se realiza primero. Como en matemáticas, puedes cambiar esto envolviendo la suma entre paréntesis:

$$(100 + 4) * 11$$

Para la resta, está el operador `-`. La división se puede hacer con el operador `/`.

Cuando los operadores aparecen juntos sin paréntesis, el orden en que se aplican se determina por la *precedencia* de los operadores. El ejemplo muestra que la multiplicación se realiza antes que la suma. El operador `/` tiene la misma precedencia que `*`. Igualmente, `+` y `-` tienen la misma precedencia. Cuando varios operadores con la misma precedencia aparecen uno al lado del otro, como en `1 - 2 + 1`, se aplican de izquierda a derecha: `(1 - 2) + 1`.

No te preocupes demasiado por estas reglas de precedencia. Cuando tengas dudas, simplemente agrega paréntesis.

Hay un operador aritmético más, que quizás no reconozcas de inmediato. El símbolo `%` se utiliza para representar la operación de *residuo*. `X % Y` es el residuo de dividir `X` por `Y`. Por ejemplo, `314 % 100` produce 14, y `144 % 12` da 0. La precedencia del operador de residuo es la misma que la de multiplicación y división. También verás a menudo a este operador referido como *módulo*.

NÚMEROS ESPECIALES

Hay tres valores especiales en JavaScript que se consideran números pero no se comportan como números normales. Los dos primeros

son Infinity y -Infinity, que representan el infinito positivo y negativo. Infinity - 1 sigue siendo Infinity, y así sucesivamente. Sin embargo, no confíes demasiado en los cálculos basados en infinito. No es matemáticamente sólido y rápidamente te llevará al siguiente número especial: NaN.

NaN significa “no es un número”, aunque es un valor del tipo numérico. Obtendrás este resultado cuando, por ejemplo, intentes calcular $0 / 0$ (cero dividido por cero), Infinity - Infinity, u cualquier otra operación numérica que no produzca un resultado significativo.

CADENAS

El siguiente tipo de dato básico es la *cadena*. Las cadenas se utilizan para representar texto. Se escriben encerrando su contenido entre comillas.

```
`En el mar`  
"Acostado en el océano"  
'Flotando en el océano'
```

Puedes usar comillas simples, comillas dobles o acentos graves para marcar las cadenas, siempre y cuando las comillas al principio y al final de la cadena coincidan.

Puedes poner casi cualquier cosa entre comillas para que JavaScript genere un valor de cadena a partir de ello. Pero algunos caracteres son más difíciles. Puedes imaginar lo complicado que sería poner comillas entre comillas, ya que parecerían el final de la cadena.

Saltos de línea (los caracteres que obtienes al presionar ENTER) solo se pueden incluir cuando la cadena está entre acentos graves (`).

Para poder incluir dichos caracteres en una cadena, se utiliza la siguiente notación: una barra invertida (\) dentro de un texto entre comillas indica que el carácter posterior tiene un significado especial. Esto se llama *escapar* el carácter. Una comilla que va precedida por una barra invertida no finalizará la cadena, sino que formará parte de ella. Cuando un carácter `n` aparece después de una barra invertida, se interpreta como un salto de línea. De manera similar, un `t` después de una barra invertida significa un carácter de tabulación. Toma la siguiente cadena:

```
"Esta es la primera línea\nY esta es la segunda"
```

Este es el texto real de esa cadena:

```
Esta es la primera línea  
Y esta es la segunda
```

Por supuesto, hay situaciones en las que deseas que una barra invertida en una cadena sea simplemente una barra invertida, no un código especial. Si dos barras invertidas van seguidas, se colapsarán juntas y solo quedará una en el valor de cadena resultante. Así es como se puede expresar la cadena “*Un carácter de nueva línea se escribe como "\n" .*”:

```
"Un carácter de nueva línea se escribe como "\\n\"."
```

Las cadenas también deben ser modeladas como una serie de bits para poder existir dentro de la computadora. La forma en que JavaScript lo hace se basa en el estándar *Unicode*. Este estándar asigna un número a prácticamente cada carácter que puedas necesitar, incluidos los caracteres griegos, árabes, japoneses,

armenios, y así sucesivamente. Si tenemos un número para cada carácter, una cadena puede ser descrita por una secuencia de números. Y eso es lo que hace JavaScript.

Sin embargo, hay una complicación: la representación de JavaScript utiliza 16 bits por elemento de cadena, lo que puede describir hasta 2^{16} caracteres diferentes. Sin embargo, Unicode define más caracteres que eso —aproximadamente el doble, en este momento. Por lo tanto, algunos caracteres, como muchos emoji, ocupan dos “posiciones de caracteres” en las cadenas de JavaScript. Volveremos a esto en [Capítulo 5](#).

Las cadenas no se pueden dividir, multiplicar o restar. El operador `+` se puede usar en ellas, no para sumar, sino para *concatenar* —unir dos cadenas. La siguiente línea producirá la cadena "concatenar":

```
"con" + "cat" + "e" + "nate"
```

Los valores de cadena tienen una serie de funciones asociadas (*métodos*) que se pueden utilizar para realizar otras operaciones con ellos. Hablaré más sobre esto en [Capítulo 4](#).

Las cadenas escritas con comillas simples o dobles se comportan de manera muy similar, la única diferencia radica en qué tipo de comilla necesitas escapar dentro de ellas. Las cadenas entre acentos graves, generalmente llamadas *template literals*, pueden hacer algunas cosas más. Aparte de poder abarcar varias líneas, también pueden incrustar otros valores.

```
`la mitad de 100 es ${100 / 2}`
```

Cuando escribes algo dentro de `${}` en una plantilla literal, su resultado se calculará, se convertirá en una cadena y se incluirá en esa posición. Este ejemplo produce “*la mitad de 100 es 50*”.

OPERADORES UNARIOS

No todos los operadores son símbolos. Algunos se escriben como palabras. Un ejemplo es el operador `typeof`, que produce un valor de cadena que indica el tipo del valor que le proporcionas.

```
console.log(typeof 4.5)
// → number
console.log(typeof "x")
// → string
```

Utilizaremos `console.log` en ejemplos de código para indicar que queremos ver el resultado de evaluar algo. Más sobre eso en el [próximo capítulo](#).

Los otros operadores mostrados hasta ahora en este capítulo operaron sobre dos valores, pero `typeof` toma solo uno. Los operadores que utilizan dos valores se llaman operadores *binarios*, mientras que aquellos que toman uno se llaman operadores *unarios*. El operador menos se puede usar tanto como un operador binario como un operador unario.

```
console.log(- (10 - 2))
// → -8
```

VALORES BOOLEANOS

A menudo es útil tener un valor que distinga solo entre dos posibilidades, como “sí” y “no” o “encendido” y “apagado”. Para este

propósito, JavaScript tiene un tipo *Booleano*, que tiene solo dos valores, true y false, escritos como esas palabras.

COMPARACIÓN

Aquí hay una forma de producir valores booleanos:

```
console.log(3 > 2)
// → true
console.log(3 < 2)
// → false
```

Los signos > y < son símbolos tradicionales para “es mayor que” y “es menor que”, respectivamente. Son operadores binarios. Aplicarlos da como resultado un valor booleano que indica si son verdaderos en este caso.

Las cadenas se pueden comparar de la misma manera:

```
console.log("Aardvark" < "Zoroaster")
// → true
```

La forma en que se ordenan las cadenas es aproximadamente alfabética pero no es realmente lo que esperarías ver en un diccionario: las letras mayúsculas son siempre “menores” que las minúsculas, por lo que "Z" < "a", y los caracteres no alfabéticos (!, -, y así sucesivamente) también se incluyen en la ordenación. Al comparar cadenas, JavaScript recorre los caracteres de izquierda a derecha, comparando los códigos Unicode uno por uno.

Otros operadores similares son >= (mayor o igual que), <= (menor o igual que), == (igual a), y != (no igual a).

```
console.log("Granate" !== "Rubí")  
// → true  
console.log("Perla" === "Amatista")  
// → false
```

Solo hay un valor en JavaScript que no es igual a sí mismo, y ese es NaN (“no es un número”).

```
console.log(NaN === NaN)  
// → false
```

NaN se supone que denota el resultado de un cálculo sin sentido y, como tal, no es igual al resultado de ningún otro cálculo sin sentido.

OPERADORES LÓGICOS

También hay algunas operaciones que se pueden aplicar a los propios valores Booleanos. JavaScript soporta tres operadores lógicos: *and* (y), *or* (o), y *not* (no). Estos se pueden usar para “razonar” sobre valores Booleanos.

El operador `&&` representa el *and* lógico. Es un operador binario, y su resultado es verdadero solo si ambos valores dados son verdaderos.

```
console.log(true && false)  
// → false  
console.log(true && true)  
// → true
```

El operador `||` representa el *or* lógico. Produce verdadero si cualquiera de los valores dados es verdadero.

```
console.log(false || true)  
// → true
```

```
console.log(false || false)
// → false
```

Not se escribe con un signo de exclamación (!). Es un operador unario que invierte el valor dado; `!true` produce `false` y `!false` produce `true`.

Al combinar estos operadores Booleanos con operadores aritméticos y otros operadores, no siempre es obvio cuándo se necesitan paréntesis. En la práctica, generalmente puedes avanzar sabiendo que de los operadores que hemos visto hasta ahora, `||` tiene la menor precedencia, luego viene `&&`, luego los operadores de comparación (`>`, `==`, etc.), y luego el resto. Este orden ha sido elegido de tal manera que, en expresiones típicas como la siguiente, se necesiten la menor cantidad de paréntesis posible:

```
1 + 1 == 2 && 10 * 10 > 50
```

El último operador lógico que veremos no es unario ni binario, sino *ternario*, operando en tres valores. Se escribe con un signo de interrogación y dos puntos, así:

```
console.log(true ? 1 : 2);
// → 1
console.log(false ? 1 : 2);
// → 2
```

Este se llama el operador *condicional* (o a veces simplemente *el operador ternario* ya que es el único operador de este tipo en el lenguaje). El operador usa el valor a la izquierda del signo de interrogación para decidir cuál de los otros dos valores “elegir”. Si

escribes `a ? b : c`, el resultado será `b` cuando `a` es verdadero y `c` de lo contrario.

VALORES VACÍOS

Hay dos valores especiales, escritos `null` y `undefined`, que se utilizan para denotar la ausencia de un valor *significativo*. Son valores en sí mismos, pero no llevan ninguna información. Muchas operaciones en el lenguaje que no producen un valor significativo devuelven `undefined` simplemente porque tienen que devolver *algún* valor.

La diferencia en el significado entre `undefined` y `null` es un accidente del diseño de JavaScript, y la mayoría de las veces no importa. En casos en los que realmente tienes que preocuparte por estos valores, recomiendo tratarlos como en su mayoría intercambiables.

CONVERSIÓN AUTOMÁTICA DE TIPOS

En la Introducción, mencioné que JavaScript se esfuerza por aceptar casi cualquier programa que le des, incluso programas que hacen cosas extrañas. Esto se demuestra claramente con las siguientes expresiones:

```
console.log(8 * null)
// → 0
console.log("5" - 1)
// → 4
console.log("5" + 1)
// → 51
console.log("five" * 2)
// → NaN
```



```
console.log(false == 0)
// → true
```

Cuando se aplica un operador al tipo de valor “incorrecto”, JavaScript convertirá silenciosamente ese valor al tipo que necesita, utilizando un conjunto de reglas que a menudo no son las que deseas o esperas. Esto se llama *coerción de tipos*. El `null` en la primera expresión se convierte en `0` y el `"5"` en la segunda expresión se convierte en `5` (de cadena a número). Sin embargo, en la tercera expresión, `+` intenta la concatenación de cadenas antes que la suma numérica, por lo que el `1` se convierte en `"1"` (de número a cadena).

Cuando algo que no se corresponde con un número de manera obvia (como `"five"` o `undefined`) se convierte en un número, obtienes el valor `NaN`. Más operaciones aritméticas en `NaN` siguen produciendo `NaN`, así que si te encuentras con uno de estos en un lugar inesperado, busca conversiones de tipo accidentales.

Cuando se comparan valores del mismo tipo usando el operador `==`, el resultado es fácil de predecir: deberías obtener verdadero cuando ambos valores son iguales, excepto en el caso de `NaN`. Pero cuando los tipos difieren, JavaScript utiliza un conjunto de reglas complicado y confuso para determinar qué hacer. En la mayoría de los casos, simplemente intenta convertir uno de los valores al tipo del otro valor. Sin embargo, cuando `null` o `undefined` aparece en cualquiera de los lados del operador, produce verdadero solo si ambos lados son uno de `null` o `undefined`.

```
console.log(null == undefined);
// → true
console.log(null == 0);
// → false
```

Ese comportamiento a menudo es útil. Cuando quieres probar si un valor tiene un valor real en lugar de `null` o `undefined`, puedes compararlo con `null` usando el operador `==` o `!=`.

¿Qué sucede si quieres probar si algo se refiere al valor preciso `false`? Expresiones como `0 == false` y `"" == false` también son verdaderas debido a la conversión automática de tipos. Cuando *no* desees que ocurran conversiones de tipo, hay dos operadores adicionales: `===` y `!==`. El primero prueba si un valor es *precisamente* igual al otro, y el segundo prueba si no es precisamente igual. Por lo tanto, `"" === false` es falso como se espera.

Recomiendo usar los operadores de comparación de tres caracteres defensivamente para evitar conversiones de tipo inesperadas que puedan complicarte las cosas. Pero cuando estés seguro de que los tipos en ambos lados serán los mismos, no hay problema en usar los operadores más cortos.

CORTOCIRCUITO DE OPERADORES LÓGICOS

Los operadores lógicos `&&` y `||` manejan valores de diferentes tipos de una manera peculiar. Convertirán el valor del lado izquierdo a tipo Booleano para decidir qué hacer, pero dependiendo del operador y el resultado de esa conversión, devolverán ya sea el valor original del lado izquierdo o el valor del lado derecho.

El operador `||`, por ejemplo, devolverá el valor de su izquierda cuando ese valor pueda convertirse en `true` y devolverá el valor de su derecha de lo contrario. Esto tiene el efecto esperado cuando los valores son Booleanos y hace algo análogo para valores de otros tipos.

```
console.log(null || "usuario")
// → usuario
console.log("Agnes" || "usuario")
// → Agnes
```

Podemos utilizar esta funcionalidad como una forma de utilizar un valor predeterminado. Si tienes un valor que podría estar vacío, puedes colocar `||` después de él con un valor de reemplazo. Si el valor inicial se puede convertir en `false`, obtendrás el valor de reemplazo en su lugar. Las reglas para convertir cadenas y números en valores Booleanos establecen que `0`, `NaN` y la cadena vacía (`""`) cuentan como `false`, mientras que todos los demás valores cuentan como `true`. Esto significa que `0 || -1` produce `-1`, y `"" || "!"` da como resultado `!"`.

El operador `??` se asemeja a `||`, pero devuelve el valor de la derecha solo si el de la izquierda es `null` o `undefined`, no si es algún otro valor que se pueda convertir en `false`. A menudo, este comportamiento es preferible al de `||`.

```
console.log(0 || 100);
// → 100
console.log(0 ?? 100);
// → 0
console.log(null ?? 100);
// → 100
```

El operador `&&` funciona de manera similar pero en sentido contrario. Cuando el valor a su izquierda es algo que se convierte en `false`, devuelve ese valor, y de lo contrario devuelve el valor de su derecha.

Otra propiedad importante de estos dos operadores es que la parte de su derecha se evalúa solo cuando es necesario. En el caso de `true || X`, no importa qué sea `X`, incluso si es una parte del programa que hace algo *terrible*, el resultado será `true`, y `X` nunca se evaluará. Lo mismo ocurre con `false && X`, que es `false` e ignorará `X`. Esto se llama *evaluación de cortocircuito*.

El operador condicional funciona de manera similar. De los valores segundo y tercero, solo se evalúa el que sea seleccionado.

RESUMEN

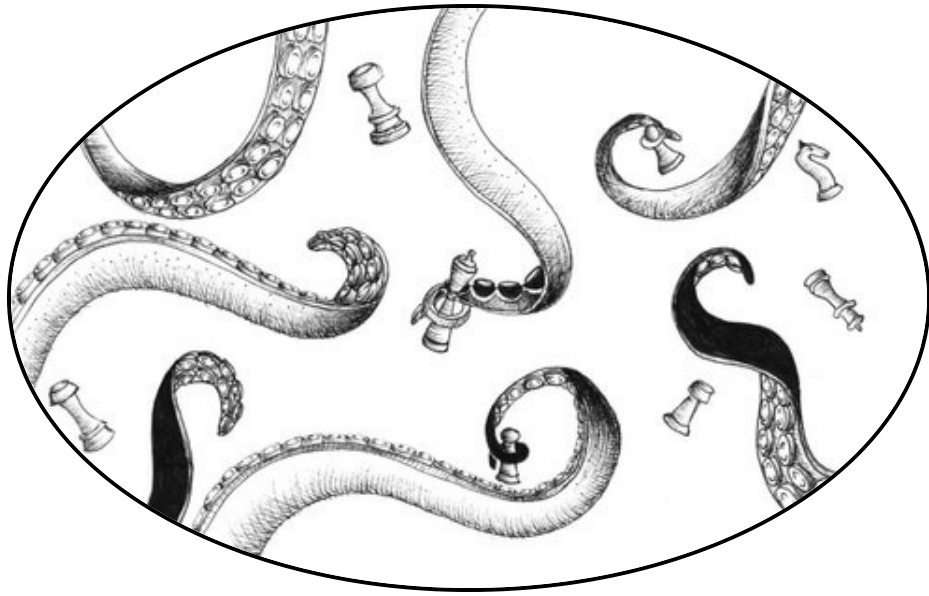
En este capítulo examinamos cuatro tipos de valores en JavaScript: números, cadenas, Booleanos y valores indefinidos. Tales valores son creados escribiendo su nombre (`true`, `null`) o valor (`13`, `"abc"`). Puedes combinar y transformar valores con operadores. Vimos operadores binarios para aritmética (`+`, `-`, `*`, `/` y `%`), concatenación de cadenas (`+`), comparación (`==`, `!=`, `===`, `!==`, `<`, `>`, `<=`, `>=`) y lógica (`&&`, `||`, `??`), así como varios operadores unarios (`-` para negar un número, `!` para negar lógicamente, y `typeof` para encontrar el tipo de un valor) y un operador ternario (`?:`) para elegir uno de dos valores basado en un tercer valor.

Esto te proporciona suficiente información para usar JavaScript como una calculadora de bolsillo, pero no mucho más. El [próximo capítulo](#) comenzará a unir estas expresiones en programas básicos.

ESTRUCTURA DEL PROGRAMA

“Y mi corazón brilla intensamente bajo mi piel diáfana y translúcida, y tienen que administrarme 10cc de JavaScript para hacerme volver. (Respondo bien a las toxinas en la sangre.) ¡Hombre, esa cosa sacará los melocotones de tus agallas!”

—_why, *Guía (conmovedora) de Ruby de Why*



En este capítulo, comenzaremos a hacer cosas que realmente pueden ser llamadas *programación*. Ampliaremos nuestro dominio del lenguaje JavaScript más allá de los sustantivos y fragmentos de oraciones que hemos visto hasta ahora, hasta el punto en que podamos expresar prosa significativa.

EXPRESIONES Y DECLARACIONES

En [Capítulo 2](#), creamos valores y aplicamos operadores a ellos para obtener nuevos valores. Crear valores de esta manera es la sustancia principal de cualquier programa JavaScript. Pero esa sustancia debe enmarcarse en una estructura más grande para ser útil. Eso es lo que cubriremos en este capítulo.

Un fragmento de código que produce un valor se llama una *expresión*. Cada valor que está escrito literalmente (como 22 o "psicoanálisis") es una expresión. Una expresión entre paréntesis también es una expresión, al igual que un operador binario aplicado a dos expresiones o un operador unario aplicado a uno.

Esto muestra parte de la belleza de una interfaz basada en lenguaje. Las expresiones pueden contener otras expresiones de manera similar a cómo las subsentencias en los idiomas humanos están anidadas: una subsentencia puede contener sus propias subsentencias, y así sucesivamente. Esto nos permite construir expresiones que describen cálculos arbitrariamente complejos.

Si una expresión corresponde a un fragmento de oración, una *declaración* de JavaScript corresponde a una oración completa. Un programa es una lista de declaraciones.

El tipo más simple de declaración es una expresión con un punto y coma al final. Este es un programa:

```
1;  
!false;
```

Sin embargo, es un programa inútil. Una expresión puede conformarse con simplemente producir un valor, que luego puede ser utilizado por el código que la contiene. Sin embargo, una declaración se mantiene por sí misma, por lo que si no afecta al mundo, es inútil. Puede mostrar algo en la pantalla, como con `console.log`, o cambiar el estado de la máquina de una manera que afectará a las declaraciones que vienen después de ella. Estos cambios se llaman *efectos secundarios*. Las declaraciones en el ejemplo anterior simplemente producen los valores `1` y `verdadero`, y luego los desechan inmediatamente. Esto no deja ninguna impresión en el mundo en absoluto. Cuando ejecutas este programa, no sucede nada observable.

En algunos casos, JavaScript te permite omitir el punto y coma al final de una declaración. En otros casos, debe estar ahí, o la próxima línea se tratará como parte de la misma declaración. Las reglas sobre cuándo se puede omitir de manera segura son algo complejas y propensas a errores. Por lo tanto, en este libro, cada declaración que necesite un punto y coma siempre recibirá uno. Te recomiendo que hagas lo mismo, al menos hasta que hayas aprendido más sobre las sutilezas de las omisiones de puntos y comas.

BINDINGS

¿Cómo mantiene un programa un estado interno? ¿Cómo recuerda las cosas? Hemos visto cómo producir nuevos valores a partir de valores antiguos, pero esto no cambia los valores antiguos, y el nuevo valor debe utilizarse inmediatamente o se disipará nuevamente. Para atrapar y retener valores, JavaScript proporciona una cosa llamada un *enlace*, o *variable*:

```
let caught = 5 * 5;
```

Eso nos da un segundo tipo de statement. La palabra especial (*keyword*) `let` indica que esta frase va a definir un enlace. Está seguida por el nombre del enlace y, si queremos darle inmediatamente un valor, por un operador `=` y una expresión.

El ejemplo crea un enlace llamado `caught` y lo utiliza para agarrar el número que se produce multiplicando 5 por 5.

Después de que se haya definido un enlace, su nombre se puede usar como una expression. El valor de esa expresión es el valor que el enlace mantiene actualmente. Aquí tienes un ejemplo:

```
let ten = 10;  
console.log(ten * ten);  
// → 100
```

Cuando un enlace apunta a un valor, eso no significa que esté atado a ese valor para siempre. El operador `=` se puede usar en cualquier momento en enlaces existentes para desconectarlos de su valor actual y hacer que apunten a uno nuevo:

```
let mood = "light";  
console.log(mood);  
// → light  
mood = "dark";  
console.log(mood);  
// → dark
```

Debes imaginarte los enlaces como tentáculos en lugar de cajas. No *contienen* valores; los *agarran*—dos enlaces pueden hacer referencia al mismo valor. Un programa solo puede acceder a los valores a los

que todavía tiene una referencia. Cuando necesitas recordar algo, o bien haces crecer un tentáculo para aferrarte a él o vuelves a conectar uno de tus tentáculos existentes a él.

Veamos otro ejemplo. Para recordar la cantidad de dólares que Luigi todavía te debe, creas un enlace. Cuando te paga \$35, le das a este enlace un nuevo valor:

```
let luigisDebt = 140;
luigisDebt = luigisDebt - 35;
console.log(luigisDebt);
// → 105
```

Cuando defines un enlace sin darle un valor, el tentáculo no tiene nada que agarrar, por lo que termina en el aire. Si solicitas el valor de un enlace vacío, obtendrás el valor `undefined`.

Una sola instrucción `let` puede definir múltiples enlaces. Las definiciones deben estar separadas por comas:

```
let one = 1, two = 2;
console.log(one + two);
// → 3
```

Las palabras `var` y `const` también se pueden usar para crear enlaces, de manera similar a `let`:

```
var name = "Ayda";
const greeting = "¡Hola ";
console.log(greeting + name);
// → ¡Hola Ayda
```

La primera de estas, `var` (abreviatura de “variable”), es la forma en que se declaraban los enlaces en JavaScript anterior a 2015, cuando

aún no existía `let`. Volveré a la forma precisa en que difiere de `let` en el [próximo capítulo](#). Por ahora, recuerda que en su mayoría hace lo mismo, pero rara vez lo usaremos en este libro porque se comporta de manera extraña en algunas situaciones.

La palabra `const` significa *constante*. Define un enlace constante, que apunta al mismo valor mientras exista. Esto es útil para enlaces que solo dan un nombre a un valor para poder referirse fácilmente a él más tarde.

NOMBRES DE ENLACES

Los nombres de enlaces pueden ser cualquier secuencia de una o más letras. Los dígitos pueden formar parte de los nombres de enlaces, `catch22` es un nombre válido, por ejemplo, pero el nombre no puede empezar con un dígito. Un nombre de enlace puede incluir signos de dólar (`$`) o subrayados (`_`), pero no otros signos de puntuación o caracteres especiales.

Palabras con un significado especial, como `let`, son *palabra clave*, y no pueden ser usadas como nombres de enlaces. También hay una serie de palabras que están “reservadas para su uso” en futuras versiones de JavaScript, las cuales tampoco se pueden usar como nombres de enlaces. La lista completa de palabras clave y palabras reservadas es bastante larga:

```
break case catch class const continue debugger default
delete do else enum export extends false finally for
function if implements import interface in instanceof let
new package private protected public return static super
switch this throw true try typeof var void while with yield
```

No te preocupes por memorizar esta lista. Cuando al crear un enlace se produce un error de sintaxis inesperado, verifica si estás intentando definir una palabra reservada.

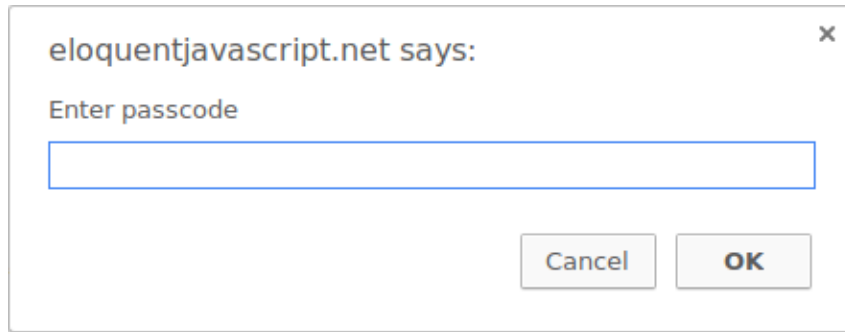
EL ENTORNO

La colección de enlaces y sus valores que existen en un momento dado se llama *entorno*. Cuando un programa se inicia, este entorno no está vacío. Siempre contiene enlaces que forman parte del lenguaje estándar, y la mayoría de las veces también tiene enlaces que proporcionan formas de interactuar con el sistema circundante. Por ejemplo, en un navegador, existen funciones para interactuar con el sitio web cargado actualmente y para leer la entrada del ratón y el teclado.

FUNCIONES

Muchos de los valores proporcionados en el entorno predeterminado tienen el tipo de *función*. Una función es un fragmento de programa envuelto en un valor. Estos valores pueden ser *aplicados* para ejecutar el programa envuelto. Por ejemplo, en un entorno de navegador, el enlace `prompt` contiene una función que muestra un pequeño cuadro de diálogo pidiendo la entrada del usuario. Se utiliza de la siguiente manera:

```
prompt("Ingrese el código de acceso");
```



Ejecutar una función se llama *invocar*, *llamar*, o *aplicar* la función. Puedes llamar una función poniendo paréntesis después de una expresión que produce un valor de función. Usualmente usarás directamente el nombre del enlace que contiene la función. Los valores entre paréntesis se le pasan al programa dentro de la función. En el ejemplo, la función `prompt` utiliza la cadena que le pasamos como el texto a mostrar en el cuadro de diálogo. Los valores dados a las funciones se llaman *argumentos*. Diferentes funciones pueden necesitar un número diferente o diferentes tipos de argumentos.

La función `prompt` no se usa mucho en la programación web moderna, principalmente porque no tienes control sobre cómo se ve el cuadro de diálogo resultante, pero puede ser útil en programas simples y experimentos.

LA FUNCIÓN `console.log`

En los ejemplos, utilicé `console.log` para mostrar valores. La mayoría de los sistemas de JavaScript (incluidos todos los navegadores web modernos y Node.js) proveen una función `console.log` que escribe sus argumentos en *algún* dispositivo de salida de texto. En los navegadores, la salida va a la consola de JavaScript. Esta parte de la interfaz del navegador está oculta por

defecto, pero la mayoría de los navegadores la abren cuando presionas F12 o, en Mac, COMANDO-OPCIÓN-I. Si eso no funciona, busca a través de los menús un elemento llamado Herramientas para Desarrolladores o similar.

Aunque los nombres de enlaces no pueden contener puntos, `console.log` tiene uno. Esto se debe a que `console.log` no es un simple enlace, sino una expresión que recupera la propiedad `log` del valor contenido por el enlace `console`. Descubriremos exactamente lo que esto significa en [Capítulo 4](#).

VALORES DE RETORNO

Mostrar un cuadro de diálogo o escribir texto en la pantalla es un efecto secundario. Muchas funciones son útiles debido a los efectos secundarios que producen. Las funciones también pueden producir valores, en cuyo caso no necesitan tener un efecto secundario para ser útiles. Por ejemplo, la función `Math.max` toma cualquier cantidad de argumentos numéricos y devuelve el mayor:

```
console.log(Math.max(2, 4));  
// → 4
```

Cuando una función produce un valor, se dice que *retorna* ese valor. Cualquier cosa que produzca un valor es una expresión en JavaScript, lo que significa que las llamadas a funciones se pueden utilizar dentro de expresiones más grandes. En el siguiente código, una llamada a `Math.min`, que es lo opuesto a `Math.max`, se usa como parte de una expresión de suma:

```
console.log(Math.min(2, 4) + 100);  
// → 102
```

[Capítulo 3](#) explicará cómo escribir tus propias funciones.

CONTROL DE FLUJO

Cuando tu programa contiene más de una sentencia, las sentencias se ejecutan como si fueran una historia, de arriba hacia abajo. Por ejemplo, el siguiente programa tiene dos sentencias. La primera le pide al usuario un número, y la segunda, que se ejecuta después de la primera, muestra el cuadrado de ese número:

```
let elNumero = Number(prompt("Elige un número"));
console.log("Tu número es la raíz cuadrada de " +
            elNumero * elNumero);
```

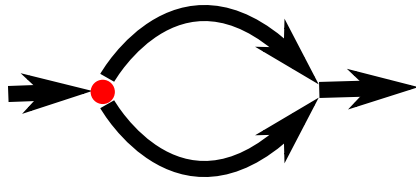
La función `Number` convierte un valor a un número. Necesitamos esa conversión porque el resultado de `prompt` es un valor de tipo `string`, y queremos un número. Hay funciones similares llamadas `String` y `Boolean` que convierten valores a esos tipos.

Aquí está la representación esquemática bastante trivial del flujo de control en línea recta:



EJECUCIÓN CONDICIONAL

No todos los programas son caminos rectos. Podríamos, por ejemplo, querer crear una carretera ramificada donde el programa tome la rama adecuada basada en la situación en cuestión. Esto se llama *ejecución condicional*.



La ejecución condicional se crea con la palabra clave `if` en JavaScript. En el caso simple, queremos que cierto código se ejecute si, y solo si, una cierta condición es verdadera. Por ejemplo, podríamos querer mostrar el cuadrado de la entrada solo si la entrada es realmente un número:

```
let elNumero = Number(prompt("Elige un número"));
if (!Number.isNaN(elNumero)) {
    console.log("Tu número es la raíz cuadrada de " +
                elNumero * elNumero);
}
```

Con esta modificación, si introduces “loro”, no se mostrará ninguna salida.

La palabra clave `if` ejecuta o salta una sentencia dependiendo del valor de una expresión booleana. La expresión de decisión se escribe después de la palabra clave, entre paréntesis, seguida de la sentencia a ejecutar.

La función `Number.isNaN` es una función estándar de JavaScript que devuelve `true` solo si el argumento que se le pasa es `NaN`. La función `Number` devuelve `NaN` cuando le das una cadena que no representa un número válido. Por lo tanto, la condición se traduce a “a menos que `elNumero` no sea un número, haz esto”.

La sentencia después del `if` está envuelta entre llaves (`{` y `}`) en este ejemplo. Las llaves se pueden usar para agrupar cualquier

cantidad de sentencias en una sola sentencia, llamada un *bloque*. También podrías haber omitido en este caso, ya que contienen solo una sentencia, pero para evitar tener que pensar si son necesarias, la mayoría de los programadores de JavaScript las usan en cada sentencia envuelta de esta manera. Seguiremos principalmente esa convención en este libro, excepto por los casos ocasionales de una sola línea.

```
if (1 + 1 == 2) console.log("Es verdad");  
// → Es verdad
```

A menudo no solo tendrás código que se ejecuta cuando una condición es verdadera, sino también código que maneja el otro caso. Esta ruta alternativa está representada por la segunda flecha en el diagrama. Puedes usar la palabra clave `else`, junto con `if`, para crear dos caminos de ejecución alternativos y separados:

```
let elNumero = Number(prompt("Elige un número"));  
if (!Number.isNaN(elNumero)) {  
    console.log("Tu número es la raíz cuadrada de " +  
                elNumero * elNumero);  
} else {  
    console.log("Oye. ¿Por qué no me diste un número?");  
}
```

Si tienes más de dos caminos para elegir, puedes “encadenar” múltiples pares `if/else`. Aquí tienes un ejemplo:

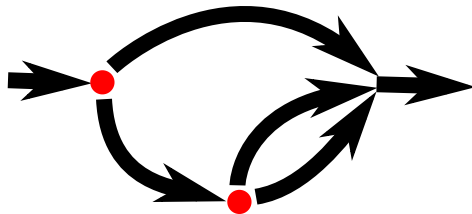
```
let num = Number(prompt("Escoge un número"));  
  
if (num < 10) {  
    console.log("Pequeño");  
} else if (num < 100) {  
    console.log("Mediano");  
} else {
```



```
    console.log("Grande");  
}
```

El programa primero comprueba si `num` es menor que 10. Si lo es, elige esa rama, muestra "Pequeño", y termina. Si no lo es, toma la rama `else`, la cual contiene a su vez otro `if`. Si la segunda condición (`< 100`) se cumple, eso significa que el número es al menos 10 pero menor que 100, y se muestra "Mediano". Si no, se elige la segunda y última rama `else`.

El esquema de este programa se ve más o menos así:



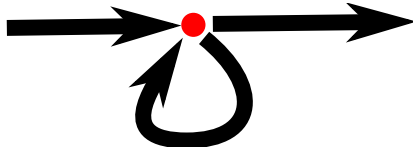
BUCLES WHILE Y DO

Considera un programa que imprime todos los números pares de 0 a 12. Una forma de escribirlo es la siguiente:

```
console.log(0);  
console.log(2);  
console.log(4);  
console.log(6);  
console.log(8);  
console.log(10);  
console.log(12);
```

Eso funciona, pero la idea de escribir un programa es hacer *menos* trabajo, no más. Si necesitáramos todos los números pares menores que 1,000, este enfoque sería inviable. Lo que necesitamos es una

manera de ejecutar un fragmento de código múltiples veces. Esta forma de control de flujo se llama *bucle*.



El control de flujo mediante bucles nos permite regresar a algún punto en el programa donde estábamos antes y repetirlo con nuestro estado de programa actual. Si combinamos esto con una variable que cuente, podemos hacer algo como esto:

```
let numero = 0;
while (numero <= 12) {
  console.log(numero);
  numero = numero + 2;
}
// → 0
// → 2
// ... etcétera
```

Un statement que comienza con la palabra clave `while` crea un bucle. La palabra `while` va seguida de una expresión entre paréntesis y luego un enunciado, similar a `if`. El bucle sigue ejecutando ese enunciado mientras la expresión produzca un valor que se convierta en `true` al convertirse a Booleano.

El enlace ‘[number](#)’ demuestra la forma en que un enlace puede seguir el progreso de un programa. Cada vez que se repite el bucle, ‘[number](#)’ obtiene un valor que es 2 más que su valor anterior. Al comienzo de cada repetición, se compara con el número 12 para decidir si el trabajo del programa ha terminado.

Como ejemplo de algo realmente útil, ahora podemos escribir un programa que calcule y muestre el valor de 2^{10} (2 elevado a la 10ª potencia). Usamos dos enlaces: uno para llevar un seguimiento de nuestro resultado y otro para contar cuántas veces hemos multiplicado este resultado por 2. El bucle comprueba si el segundo enlace ha alcanzado 10 aún y, si no, actualiza ambos enlaces.

```
let result = 1;
let counter = 0;
while (counter < 10) {
  result = result * 2;
  counter = counter + 1;
}
console.log(result);
// → 1024
```

El contador también podría haber comenzado en 1 y haber comprobado si era ≤ 10 , pero por razones que se harán evidentes en [Capítulo 4](#), es buena idea acostumbrarse a contar desde 0.

Ten en cuenta que JavaScript también tiene un operador para la exponenciación ($2^{**}10$), que usarías para calcular esto en un código real, pero eso habría arruinado el ejemplo.

Un bucle `do` es una estructura de control similar a un bucle `while`. La única diferencia radica en que un bucle `do` siempre ejecuta su cuerpo al menos una vez, y comienza a probar si debe detenerse solo después de esa primera ejecución. Para reflejar esto, la prueba aparece después del cuerpo del bucle:

```
let tuNombre;
do {
  tuNombre = prompt("¿Quién eres?");
```

```
} while (!tuNombre);  
console.log("Hola " + tuNombre);
```

Este programa te obligará a ingresar un nombre. Preguntará una y otra vez hasta que obtenga algo que no sea una cadena vacía. Aplicar el operador `!` convertirá un valor al tipo Booleano antes de negarlo, y todas las cadenas excepto `" "` se convierten en `true`. Esto significa que el bucle continúa hasta que proporciones un nombre no vacío.

SANGRADO DE CÓDIGO

En los ejemplos, he estado agregando espacios delante de las sentencias que son parte de alguna otra sentencia más grande. Estos espacios no son necesarios: la computadora aceptará el programa perfectamente sin ellos. De hecho, incluso los saltos de línea en los programas son opcionales. Podrías escribir un programa como una sola línea larga si así lo deseas.

El papel de este sangrado dentro de los bloques es hacer que la estructura del código resalte para los lectores humanos. En el código donde se abren nuevos bloques dentro de otros bloques, puede volverse difícil ver dónde termina un bloque y comienza otro. Con un sangrado adecuado, la forma visual de un programa corresponde a la forma de los bloques dentro de él. A mí me gusta usar dos espacios para cada bloque abierto, pero los gustos difieren: algunas personas usan cuatro espacios y otras usan caracteres de tabulación. Lo importante es que cada nuevo bloque agregue la misma cantidad de espacio.

```
if (false != true) {  
    console.log("Tiene sentido.");  
    if (1 < 2) {
```

```
    console.log("No hay sorpresas ahí.");  
  }  
}
```

La mayoría de los programas de edición (incluido el de este libro) ayudarán automáticamente con la sangría adecuada al escribir nuevas líneas.

BUCLÉS FOR

Muchos bucles siguen el patrón mostrado en los ejemplos de `while`. Primero se crea una variable de “contador” para rastrear el progreso del bucle. Luego viene un bucle `while`, generalmente con una expresión de prueba que verifica si el contador ha alcanzado su valor final. Al final del cuerpo del bucle, el contador se actualiza para rastrear el progreso.

Debido a que este patrón es tan común, JavaScript y lenguajes similares proporcionan una forma ligeramente más corta y completa, el bucle `for`:

```
for (let numero = 0; numero <= 12; numero = numero + 2) {  
  console.log(numero);  
}  
// → 0  
// → 2  
// ... etcétera
```

Este programa es exactamente equivalente al [anterior](#) ejemplo de impresión de números pares. La única diferencia es que todas las declaraciones relacionadas con el “estado” del bucle están agrupadas después de `for`.

Los paréntesis después de la palabra clave `for` deben contener dos punto y coma. La parte antes del primer punto y coma *inicializa* el bucle, generalmente definiendo una variable. La segunda parte es la expresión que *verifica* si el bucle debe continuar. La parte final *actualiza* el estado del bucle después de cada iteración. En la mayoría de los casos, esto es más corto y claro que un `while` tradicional.

Este es el código que calcula 2^{10} usando `for` en lugar de `while`:

```
let resultado = 1;
for (let contador = 0; contador < 10; contador = contador + 1) {
  resultado = resultado * 2;
}
console.log(resultado);
// → 1024
```

SALIENDO DE UN BUCLE

Hacer que la condición del bucle produzca `false` no es la única forma en que un bucle puede terminar. La instrucción `break` tiene el efecto de salir inmediatamente del bucle que la contiene. Su uso se demuestra en el siguiente programa, que encuentra el primer número que es mayor o igual a 20 y divisible por 7:

```
for (let actual = 20; ; actual = actual + 1) {
  if (actual % 7 == 0) {
    console.log(actual);
    break;
  }
}
// → 21
```

Usar el operador de resto (%) es una forma sencilla de comprobar si un número es divisible por otro. Si lo es, el resto de su división es cero.

La construcción `for` en el ejemplo no tiene una parte que verifique el final del bucle. Esto significa que el bucle nunca se detendrá a menos que se ejecute la instrucción `break` dentro de él.

Si eliminaras esa declaración `break` o escribieses accidentalmente una condición final que siempre produzca `true`, tu programa quedaría atrapado en un *bucle infinito*. Un programa atrapado en un bucle infinito nunca terminará de ejecutarse, lo cual suele ser algo malo.

La palabra clave `continue` es similar a `break` en que influye en el progreso de un bucle. Cuando se encuentra `continue` en el cuerpo de un bucle, el control salta fuera del cuerpo y continúa con la siguiente iteración del bucle.

ACTUALIZACIÓN CONCISA DE ENLACES

Especialmente al hacer bucles, un programa a menudo necesita “actualizar” un enlace para que contenga un valor basado en el valor anterior de ese enlace.

```
counter = counter + 1;
```

JavaScript proporciona un atajo para esto:

```
counter += 1;
```

Atajos similares funcionan para muchos otros operadores, como `result *= 2` para duplicar `result` o `counter -= 1` para contar hacia abajo.

Esto nos permite acortar aún más nuestro ejemplo de contar:

```
for (let number = 0; number <= 12; number += 2) {  
  console.log(number);  
}
```

Para `counter += 1` y `counter -= 1`, existen equivalentes aún más cortos: `counter++` y `counter--`.

DESPACHAR UN VALOR CON SWITCH

No es raro que el código luzca así:

```
if (x == "valor1") accion1();  
else if (x == "valor2") accion2();  
else if (x == "valor3") accion3();  
else accionPredeterminada();
```

Existe una construcción llamada `switch` que está destinada a expresar dicho “despacho” de una manera más directa.

Desafortunadamente, la sintaxis que JavaScript utiliza para esto (heredada de la línea de lenguajes de programación C/Java) es algo incómoda; una cadena de declaraciones `if` puede verse mejor. Aquí hay un ejemplo:

```
switch (prompt("¿Cómo está el clima?")) {  
  case "lluvioso":  
    console.log("Recuerda llevar un paraguas.");  
    break;  
  case "soleado":  
    console.log("Vístete ligero.");  
}
```



```
case "nublado":
    console.log("Sal al exterior.");
    break;
default:
    console.log("¡Tipo de clima desconocido!");
    break;
}
```

Puedes colocar cualquier cantidad de etiquetas `case` dentro del bloque abierto por `switch`. El programa comenzará a ejecutarse en la etiqueta que corresponda al valor que se le dio a `switch`, o en `default` si no se encuentra ningún valor coincidente. Continuará ejecutándose, incluso a través de otras etiquetas, hasta que alcance una declaración `break`. En algunos casos, como el caso `"soleado"` en el ejemplo, esto se puede usar para compartir algo de código entre casos (recomienda salir al exterior tanto para el clima soleado como para el nublado). Sin embargo, ten cuidado, es fácil olvidar un `break` de este tipo, lo que hará que el programa ejecute código que no deseas ejecutar.

CAPITALIZACIÓN

Los nombres de los enlaces no pueden contener espacios, sin embargo, a menudo es útil usar varias palabras para describir claramente lo que representa el enlace. Estas son básicamente tus opciones para escribir un nombre de enlace con varias palabras:

```
fuzzylittleturtle
fuzzy_little_turtle
FuzzyLittleTurtle
fuzzyLittleTurtle
```

El primer estilo puede ser difícil de leer. Personalmente me gusta más la apariencia de los guiones bajos, aunque ese estilo es un poco

difícil de escribir. Las funciones estándar de JavaScript y la mayoría de los programadores de JavaScript siguen el último estilo: capitalizan cada palabra excepto la primera. No es difícil acostumbrarse a pequeñas cosas como esa, y el código con estilos de nombrado mixtos puede resultar molesto de leer, así que seguimos esta convención.

En algunos casos, como en la función `Number`, la primera letra de un enlace también está en mayúscula. Esto se hizo para marcar esta función como un constructor. Quedará claro lo que es un constructor en [Capítulo 6](#). Por ahora, lo importante es no molestarse por esta aparente falta de consistencia.

COMENTARIOS

A menudo, el código sin formato no transmite toda la información que deseas que un programa transmita a los lectores humanos, o lo hace de una manera tan críptica que las personas podrían no entenderlo. En otros momentos, es posible que solo quieras incluir algunos pensamientos relacionados como parte de tu programa. Para eso sirven los *comentarios*.

Un comentario es un fragmento de texto que forma parte de un programa pero que es completamente ignorado por la computadora. JavaScript tiene dos formas de escribir comentarios. Para escribir un comentario de una sola línea, puedes usar dos caracteres de barra (`//`) y luego el texto del comentario después de eso:

```
let saldoCuenta = calcularSaldo(cuenta);  
// Es un hueco verde donde canta un río  
saldoCuenta.ajustar();  
// Atrapando locamente pedazos blancos en la hierba.
```

```
let informe = new Informe();  
// Donde el sol en la orgullosa montaña resuena:  
agregarAInforme(saldoCuenta, informe);  
// Es un valle pequeño, espumoso como la luz en un vaso.
```

Un comentario con `//` solo va hasta el final de la línea. Una sección de texto entre `/*` y `*/` será ignorada por completo, independientemente de si contiene saltos de línea. Esto es útil para agregar bloques de información sobre un archivo o un fragmento de programa:

```
/*  
    Encontré este número por primera vez garabateado en la parte  
    posterior de un viejo  
    cuaderno. Desde entonces, a menudo ha aparecido, mostrándose en  
    números de teléfono y números de serie de productos que he  
    comprado. Obviamente le gusto, así que he decidido quedármelo.  
*/  
const miNumero = 11213;
```

RESUMEN

Ahora sabes que un programa está construido a partir de declaraciones, que a veces contienen más declaraciones. Las declaraciones tienden a contener expresiones, que a su vez pueden estar construidas a partir de expresiones más pequeñas. Poner declaraciones una después de la otra te da un programa que se ejecuta de arriba hacia abajo. Puedes introducir alteraciones en el flujo de control usando declaraciones condicionales (`if`, `else` y `switch`) y bucles (`while`, `do` y `for`).

Las uniones se pueden usar para guardar fragmentos de datos bajo un nombre, y son útiles para hacer un seguimiento del estado en tu programa. El entorno es el conjunto de uniones que están definidas.

Los sistemas de JavaScript siempre colocan varias uniones estándar útiles en tu entorno.

Las funciones son valores especiales que encapsulan un fragmento de programa. Puedes invocarlas escribiendo `nombreDeFuncion(argumento1, argumento2)`. Dicha llamada a función es una expresión y puede producir un valor.

EJERCICIOS

Si no estás seguro de cómo probar tus soluciones a los ejercicios, consulta la [Introducción](#).

Cada ejercicio comienza con una descripción del problema. Lee esta descripción e intenta resolver el ejercicio. Si encuentras problemas, considera leer las pistas al [final del libro](#). Puedes encontrar soluciones completas a los ejercicios en línea en <https://eloquentjavascript.net/code>. Si deseas aprender algo de los ejercicios, te recomiendo mirar las soluciones solo después de haber resuelto el ejercicio, o al menos después de haberlo intentado lo suficiente como para tener un ligero dolor de cabeza.

HACIENDO UN TRIÁNGULO CON BUCLES

Escribe un bucle que realice siete llamadas a `console.log` para mostrar el siguiente triángulo:

```
#  
##  
###  
####  
#####
```

```
#####  
#####
```

Puede ser útil saber que puedes encontrar la longitud de una cadena escribiendo `.length` después de ella.

```
let abc = "abc";  
console.log(abc.length);  
// → 3
```

FIZZBUZZ

Escribe un programa que use `console.log` para imprimir todos los números del 1 al 100, con dos excepciones. Para los números divisibles por 3, imprime "Fizz" en lugar del número, y para los números divisibles por 5 (y no por 3), imprime "Buzz" en su lugar.

Cuando tengas eso funcionando, modifica tu programa para imprimir "FizzBuzz" para los números que son divisibles por 3 y 5 (y sigue imprimiendo "Fizz" o "Buzz" para los números que son divisibles solo por uno de esos).

(Esto es en realidad una pregunta de entrevista que se ha afirmado que elimina a un porcentaje significativo de candidatos a programadores. Entonces, si lo resolviste, tu valor en el mercado laboral acaba de aumentar.)

TABLERO DE AJEDREZ

Escribe un programa que cree una cadena que represente un tablero de 8x8, usando caracteres de salto de línea para separar las líneas. En cada posición del tablero hay un carácter de espacio o un carácter "#". Los caracteres deben formar un tablero de ajedrez.

Al pasar esta cadena a `console.log` debería mostrar algo como esto:

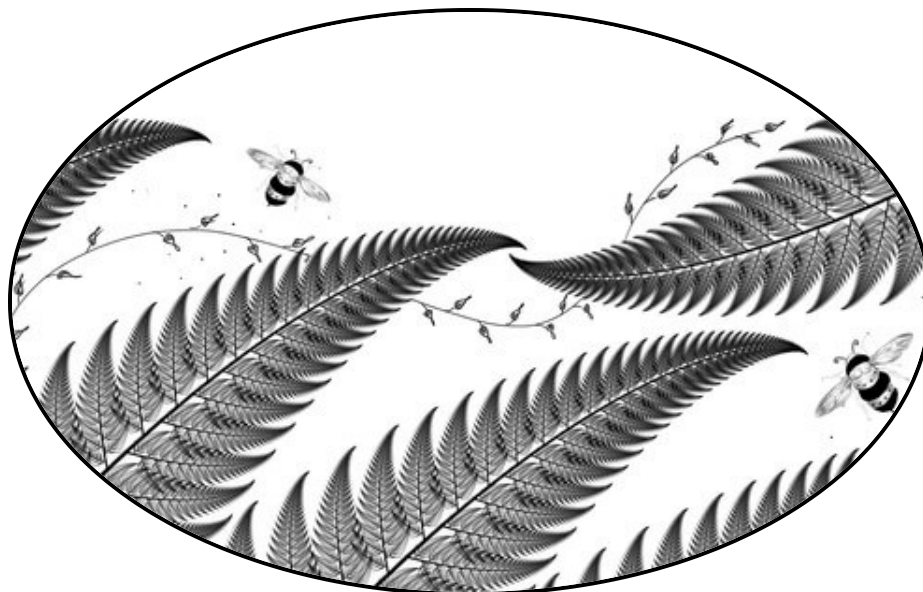
```
# # # #  
# # # #  
# # # #  
# # # #  
# # # #  
# # # #  
# # # #  
# # # #
```

Cuando tengas un programa que genere este patrón, define una variable `size = 8` y cambia el programa para que funcione para cualquier `size`, generando un tablero con el ancho y alto dados.

FUNCIONES

“La gente piensa que la informática es el arte de los genios, pero la realidad actual es la opuesta, simplemente muchas personas haciendo cosas que se construyen unas sobre otras, como un muro de mini piedras.”

—Donald Knuth



Las funciones son una de las herramientas más centrales en la programación en JavaScript. El concepto de envolver un fragmento de programa en un valor tiene muchos usos. Nos proporciona una manera de estructurar programas más grandes, de reducir la repetición, de asociar nombres con subprogramas y de aislar estos subprogramas entre sí.

La aplicación más evidente de las funciones es definir nuevos vocabulario. Crear nuevas palabras en prosa suele ser de mal estilo,

pero en la programación es indispensable.

Los hablantes de inglés adultos típicos tienen alrededor de 20,000 palabras en su vocabulario. Pocas lenguajes de programación vienen con 20,000 comandos incorporados. Y el vocabulario que *está* disponible tiende a estar más precisamente definido, y por lo tanto menos flexible, que en el lenguaje humano. Por lo tanto, *tenemos* que introducir nuevas palabras para evitar la verbosidad excesiva.

DEFINIR UNA FUNCIÓN

Una definición de función es una vinculación regular donde el valor de la vinculación es una función. Por ejemplo, este código define `square` para que se refiera a una función que produce el cuadrado de un número dado:

```
const square = function(x) {  
  return x * x;  
};  
  
console.log(square(12));  
// → 144
```

Una función se crea con una expresión que comienza con la palabra clave `function`. Las funciones tienen un conjunto de *parámetros* (en este caso, solo `x`) y un *cuerpo*, que contiene las declaraciones que se ejecutarán cuando se llame a la función. El cuerpo de una función creada de esta manera siempre debe estar envuelto entre llaves, incluso cuando consiste en una única declaración.

Una función puede tener varios parámetros o ninguno en absoluto. En el siguiente ejemplo, `makeNoise` no enumera nombres de

parámetros, mientras que `roundTo` (que redondea `n` al múltiplo más cercano de `step`) enumera dos:

```
const makeNoise = function() {  
  console.log("¡Pling!");  
};  
  
makeNoise();  
// → ¡Pling!  
  
const roundTo = function(n, step) {  
  let resto = n % step;  
  return n - resto + (resto < step / 2 ? 0 : step);  
};  
  
console.log(roundTo(23, 10));  
// → 20
```

Algunas funciones, como `roundTo` y `square`, producen un valor, y otras no, como `makeNoise`, cuyo único resultado es un efecto secundario. Una instrucción `return` determina el valor que devuelve la función. Cuando el control llega a una instrucción de ese tipo, salta inmediatamente fuera de la función actual y le da el valor devuelto al código que llamó a la función. Una palabra clave `return` sin una expresión después de ella hará que la función devuelva `undefined`. Las funciones que no tienen ninguna instrucción `return` en absoluto, como `makeNoise`, devuelven igualmente `undefined`.

Los parámetros de una función se comportan como ligaduras regulares, pero sus valores iniciales son dados por el *llamador* de la función, no por el código en la función en sí misma.

LIGADURAS Y ÁMBITOS

Cada ligadura tiene un *ámbito*, que es la parte del programa en la que la ligadura es visible. Para las ligaduras definidas fuera de cualquier función, bloque o módulo (ver [Capítulo ?](#)), el ámbito es todo el programa—puedes hacer referencia a esas ligaduras donde quieras. Estas se llaman *globales*.

Las ligaduras creadas para los parámetros de una función o declaradas dentro de una función solo pueden ser referenciadas en esa función, por lo que se conocen como ligaduras *locales*. Cada vez que se llama a la función, se crean nuevas instancias de estas ligaduras. Esto proporciona cierto aislamiento entre funciones—cada llamada a función actúa en su propio pequeño mundo (su entorno local) y a menudo se puede entender sin saber mucho sobre lo que está sucediendo en el entorno global.

Las ligaduras declaradas con `let` y `const` en realidad son locales al *bloque* en el que se declaran, por lo que si creas una de esas dentro de un bucle, el código antes y después del bucle no puede “verla”. En JavaScript anterior a 2015, solo las funciones creaban nuevos ámbitos, por lo que las ligaduras de estilo antiguo, creadas con la palabra clave `var`, son visibles en toda la función en la que aparecen—o en todo el ámbito global, si no están dentro de una función.

```
let x = 10;    // global
if (true) {
  let y = 20; // local al bloque
  var z = 30; // también global
}
```

Cada ámbito puede “mirar hacia afuera” al ámbito que lo rodea, por lo que `x` es visible dentro del bloque en el ejemplo. La excepción es cuando múltiples ligaduras tienen el mismo nombre—en ese caso, el

código solo puede ver la más interna. Por ejemplo, cuando el código dentro de la función `halve` hace referencia a `n`, está viendo su *propio* `n`, no el `n` global.

```
const halve = function(n) {  
  return n / 2;  
};  
  
let n = 10;  
console.log(halve(100));  
// → 50  
console.log(n);  
// → 10
```

ÁMBITO ANIDADO

JavaScript distingue no solo entre ligaduras globales y locales. Bloques y funciones pueden ser creados dentro de otros bloques y funciones, produciendo múltiples grados de localidad.

Por ejemplo, esta función—que muestra los ingredientes necesarios para hacer un lote de hummus—tiene otra función dentro de ella:

```
const hummus = function(factor) {  
  const ingredient = function(amount, unit, name) {  
    let ingredientAmount = amount * factor;  
    if (ingredientAmount > 1) {  
      unit += "s";  
    }  
    console.log(`${ingredientAmount} ${unit} ${name}`);  
  };  
  ingredient(1, "lata", "garbanzos");  
  ingredient(0.25, "taza", "tahini");  
  ingredient(0.25, "taza", "jugo de limón");  
  ingredient(1, "diente", "ajo");  
  ingredient(2, "cucharada", "aceite de oliva");  
};
```

```
    ingredient(0.5, "cucharadita", "comino");  
};
```

El código dentro de la función `ingredient` puede ver el enlace `factor` desde la función exterior, pero sus enlaces locales, como `unit` o `ingredientAmount`, no son visibles en la función exterior.

El conjunto de enlaces visibles dentro de un bloque está determinado por el lugar de ese bloque en el texto del programa. Cada bloque local también puede ver todos los bloques locales que lo contienen, y todos los bloques pueden ver el bloque global. Este enfoque de visibilidad de enlaces se llama *lexicografía*.

FUNCIONES COMO VALORES

Un enlace de función generalmente simplemente actúa como un nombre para una parte específica del programa. Este enlace se define una vez y nunca se cambia. Esto hace que sea fácil confundir la función y su nombre.

Pero los dos son diferentes. Un valor de función puede hacer todas las cosas que pueden hacer otros valores: se puede utilizar en expresiones arbitrarias, no solo llamarlo. Es posible almacenar un valor de función en un nuevo enlace, pasarlo como argumento a una función, etc. De manera similar, un enlace que contiene una función sigue siendo solo un enlace regular y, si no es constante, se le puede asignar un nuevo valor, así:

```
let launchMissiles = function() {  
    missileSystem.launch("now");  
};  
if (safeMode) {
```

```
    launchMissiles = function() { /* no hacer nada */ };  
}
```

En [Capítulo 5](#), discutiremos las cosas interesantes que podemos hacer al pasar valores de función a otras funciones.

NOTACIÓN DE DECLARACIÓN

Hay una manera ligeramente más corta de crear un enlace de función. Cuando se utiliza la palabra clave `function` al inicio de una declaración, funciona de manera diferente:

```
function square(x) {  
    return x * x;  
}
```

Esta es una función *declarativa*. La declaración define el enlace `square` y lo apunta a la función dada. Es un poco más fácil de escribir y no requiere un punto y coma después de la función.

Hay una sutileza con esta forma de definición de función.

```
console.log("El futuro dice:", future());  
  
function future() {  
    return "Nunca tendrás autos voladores";  
}
```

El código anterior funciona, incluso aunque la función esté definida *debajo* del código que la usa. Las declaraciones de función no forman parte del flujo de control regular de arriba hacia abajo.

Conceptualmente se mueven al principio de su alcance y pueden ser utilizadas por todo el código en ese alcance. A veces esto es útil porque ofrece la libertad de ordenar el código de una manera que

parezca más clara, sin tener que preocuparse por definir todas las funciones antes de que se utilicen.

FUNCIONES DE FLECHA

Hay una tercera notación para funciones, que se ve muy diferente de las otras. En lugar de la palabra clave `function`, utiliza una flecha (`=>`) compuesta por un signo igual y un caracter mayor que (no confundir con el operador mayor o igual, que se escribe `>=`):

```
const roundTo = (n, step) => {  
  let remainder = n % step;  
  return n - remainder + (remainder < step / 2 ? 0 : step);  
};
```

La flecha viene *después* de la lista de parámetros y es seguida por el cuerpo de la función. Expresa algo así como “esta entrada (los parámetros) produce este resultado (el cuerpo)”.

Cuando solo hay un nombre de parámetro, puedes omitir los paréntesis alrededor de la lista de parámetros. Si el cuerpo es una sola expresión, en lugar de un bloque entre llaves, esa expresión será devuelta por la función. Por lo tanto, estas dos definiciones de `exponente` hacen lo mismo:

```
const exponente1 = (x) => { return x * x; };  
const exponente2 = x => x * x;
```

Cuando una función de flecha no tiene parámetros en absoluto, su lista de parámetros es simplemente un conjunto vacío de paréntesis.

```
const cuerno = () => {  
  console.log("Toot");  
};
```

No hay una razón profunda para tener tanto funciones de flecha como expresiones `function` en el lenguaje. Aparte de un detalle menor, que discutiremos en el [Capítulo ?](#), hacen lo mismo. Las funciones de flecha se agregaron en 2015, principalmente para hacer posible escribir expresiones de función pequeñas de una manera menos verbosa. Las usaremos a menudo en [Capítulo ?](orden superior).

LA PILA DE LLAMADAS

La forma en que el control fluye a través de las funciones es un tanto complicada. Echemos un vistazo más de cerca. Aquí hay un programa simple que realiza algunas llamadas de función:

```
function saludar(quién) {  
  console.log("Hola " + quién);  
}  
saludar("Harry");  
console.log("Adiós");
```

Una ejecución de este programa va más o menos así: la llamada a `saludar` hace que el control salte al inicio de esa función (línea 2). La función llama a `console.log`, que toma el control, hace su trabajo, y luego devuelve el control a la línea 2. Allí, llega al final de la función `saludar`, por lo que regresa al lugar que la llamó, línea 4. La línea siguiente llama a `console.log` nuevamente. Después de ese retorno, el programa llega a su fin.

Podríamos mostrar el flujo de control esquemáticamente de esta manera:

```
no en función
  en saludar
    en console.log
  en saludar
no en función
  en console.log
no en función
```

Dado que una función tiene que regresar al lugar que la llamó cuando termina, la computadora debe recordar el contexto desde el cual se realizó la llamada. En un caso, `console.log` tiene que regresar a la función `saludar` cuando haya terminado. En el otro caso, regresa al final del programa.

El lugar donde la computadora almacena este contexto es la *pila de llamadas*. Cada vez que se llama a una función, el contexto actual se almacena en la parte superior de esta pila. Cuando una función devuelve, elimina el contexto superior de la pila y usa ese contexto para continuar la ejecución.

Almacenar esta pila requiere espacio en la memoria de la computadora. Cuando la pila crece demasiado, la computadora fallará con un mensaje como “sin espacio en la pila” o “demasiada recursividad”. El siguiente código ilustra esto al hacerle a la computadora una pregunta realmente difícil que causa un vaivén infinito entre dos funciones. O más bien, *sería* infinito, si la computadora tuviera una pila infinita. Como no la tiene, nos quedaremos sin espacio o “reventaremos la pila”.

```
function chicken() {
  return egg();
}
function egg() {
  return chicken();
}
```



```
}  
console.log(chicken() + " salió primero.");  
// → ??
```

ARGUMENTOS OPCIONALES

El siguiente código está permitido y se ejecuta sin ningún problema:

```
function square(x) { return x * x; }  
console.log(square(4, true, "erizo"));  
// → 16
```

Hemos definido `square` con solo un parámetro. Sin embargo, cuando lo llamamos con tres, el lenguaje no se queja. Ignora los argumentos adicionales y calcula el cuadrado del primero.

JavaScript es extremadamente flexible en cuanto al número de argumentos que puedes pasar a una función. Si pasas demasiados, los extras son ignorados. Si pasas muy pocos, los parámetros faltantes se les asigna el valor `undefined`.

El inconveniente de esto es que es posible —incluso probable— que pases accidentalmente el número incorrecto de argumentos a las funciones. Y nadie te dirá nada al respecto. La ventaja es que puedes utilizar este comportamiento para permitir que una función sea llamada con diferentes números de argumentos. Por ejemplo, esta función `minus` intenta imitar al operador `-` actuando sobre uno o dos argumentos:

```
function minus(a, b) {  
  if (b === undefined) return -a;  
  else return a - b;  
}
```

```
console.log(minus(10));  
// → -10  
console.log(minus(10, 5));  
// → 5
```

Si escribes un operador `=` después de un parámetro, seguido de una expresión, el valor de esa expresión reemplazará al argumento cuando no se le dé. Por ejemplo, esta versión de `roundTo` hace que su segundo argumento sea opcional. Si no lo proporcionas o pasas el valor `undefined`, por defecto será uno:

```
function roundTo(n, step = 1) {  
  let remainder = n % step;  
  return n - remainder + (remainder < step / 2 ? 0 : step);  
};  
  
console.log(roundTo(4.5));  
// → 5  
console.log(roundTo(4.5, 2));  
// → 4
```

[El próximo capítulo](#) introducirá una forma en que un cuerpo de función puede acceder a la lista completa de argumentos que se le pasaron. Esto es útil porque le permite a una función aceptar cualquier número de argumentos. Por ejemplo, `console.log` lo hace, mostrando todos los valores que se le dan:

```
console.log("C", "0", 2);  
// → C 0 2
```

CLAUSURA

La capacidad de tratar las funciones como valores, combinada con el hecho de que las vinculaciones locales se recrean cada vez que se llama a una función, plantea una pregunta interesante: ¿qué sucede

con las vinculaciones locales cuando la llamada a la función que las creó ya no está activa? El siguiente código muestra un ejemplo de esto. Define una función, `wrapValue`, que crea un enlace local. Luego devuelve una función que accede y devuelve este enlace local:

```
function wrapValue(n) {  
  let local = n;  
  return () => local;  
}  
  
let wrap1 = wrapValue(1);  
let wrap2 = wrapValue(2);  
console.log(wrap1());  
// → 1  
console.log(wrap2());  
// → 2
```

Esto está permitido y funciona como esperarías: ambas instancias del enlace aún pueden accederse. Esta situación es una buena demostración de que los enlaces locales se crean nuevamente para cada llamada, y las diferentes llamadas no afectan los enlaces locales de los demás.

Esta característica, poder hacer referencia a una instancia específica de un enlace local en un ámbito superior, se llama *clausura*. Una función que hace referencia a enlaces de ámbitos locales a su alrededor se llama *una* clausura. Este comportamiento no solo te libera de tener que preocuparte por la vida útil de los enlaces, sino que también hace posible usar valores de función de formas creativas.

Con un ligero cambio, podemos convertir el ejemplo anterior en una forma de crear funciones que multiplican por una cantidad arbitraria:

```
function multiplier(factor) {  
  return number => number * factor;  
}  
  
let twice = multiplier(2);  
console.log(twice(5));  
// → 10
```

El enlace explícito `local` del ejemplo `wrapValue` realmente no es necesario, ya que un parámetro es en sí mismo un enlace local.

Pensar en programas de esta manera requiere algo de práctica. Un buen modelo mental es pensar en los valores de función como que contienen tanto el código en su cuerpo como el entorno en el que fueron creados. Cuando se llama, el cuerpo de la función ve el entorno en el que fue creado, no el entorno en el que se llama.

En el ejemplo anterior, se llama a `multiplier` y crea un entorno en el que su parámetro `factor` está vinculado a 2. El valor de función que devuelve, que se almacena en `twice`, recuerda este entorno para que cuando se llame, multiplique su argumento por 2.

RECURSIÓN

Es perfectamente válido que una función se llame a sí misma, siempre y cuando no lo haga tan a menudo que desborde la pila. Una función que se llama a sí misma se llama *recursiva*. La recursión permite que algunas funciones se escriban de una manera diferente. Toma, por ejemplo, esta función `power`, que hace lo mismo que el operador `**` (exponenciación):

```
function power(base, exponent) {  
  if (exponent == 0) {  
    return 1;  
  }
```

```
    } else {  
        return base * power(base, exponent - 1);  
    }  
}  
  
console.log(power(2, 3));  
// → 8
```

Esto se asemeja bastante a la forma en que los matemáticos definen la exponenciación y describe el concepto de manera más clara que el bucle que usamos en [Capítulo ?](#). La función se llama a sí misma varias veces con exponentes cada vez más pequeños para lograr la multiplicación repetida.

Sin embargo, esta implementación tiene un problema: en implementaciones típicas de JavaScript, es aproximadamente tres veces más lenta que una versión que utiliza un `for` loop. Recorrer un simple bucle suele ser más económico que llamar a una función múltiples veces.

El dilema de velocidad versus elegancia es interesante. Se puede ver como una especie de continuo entre amigabilidad humana y amigabilidad de máquina. Casi cualquier programa puede ser acelerado haciendo que sea más extenso y complicado. El programador debe encontrar un equilibrio apropiado.

En el caso de la función `potencia`, una versión poco elegante (con bucles) sigue siendo bastante simple y fácil de leer. No tiene mucho sentido reemplazarla con una función recursiva. Sin embargo, a menudo un programa trata con conceptos tan complejos que renunciar a algo de eficiencia para hacer que el programa sea más directo es útil.

Preocuparse por la eficiencia puede ser una distracción. Es otro factor que complica el diseño del programa y cuando estás haciendo algo que ya es difícil, ese extra en lo que preocuparse puede llegar a ser paralizante.

Por lo tanto, generalmente deberías comenzar escribiendo algo que sea correcto y fácil de entender. Si te preocupa que sea demasiado lento—lo cual suele ser raro, ya que la mayoría del código simplemente no se ejecuta lo suficiente como para tomar una cantidad significativa de tiempo—puedes medir después y mejorarlo si es necesario.

La recursión no siempre es simplemente una alternativa ineficiente a los bucles. Algunos problemas realmente son más fáciles de resolver con recursión que con bucles. Con mayor frecuencia, estos son problemas que requieren explorar o procesar varias “ramas”, cada una de las cuales podría ramificarse nuevamente en aún más ramas.

Considera este rompecabezas: al comenzar desde el número 1 y repetidamente sumar 5 o multiplicar por 3, se puede producir un conjunto infinito de números. ¿Cómo escribirías una función que, dado un número, intente encontrar una secuencia de tales sumas y multiplicaciones que produzcan ese número? Por ejemplo, el número 13 podría alcanzarse al multiplicar por 3 y luego sumar 5 dos veces, mientras que el número 15 no podría alcanzarse en absoluto.

Aquí tienes una solución recursiva:

```
function findSolution(objetivo) {  
  function find(actual, historial) {  
    if (actual === objetivo) {  
      return historial;  
    }  
  }  
}
```

```

    } else if (actual > objetivo) {
        return null;
    } else {
        return find(actual + 5, `${historial} + 5`) ??
            find(actual * 3, `${historial} * 3`);
    }
}
return find(1, "1");
}

console.log(findSolution(24));
// → (((1 * 3) + 5) * 3)

```

Ten en cuenta que este programa no necesariamente encuentra la secuencia de operaciones más *corta*. Se conforma con encontrar cualquier secuencia en absoluto.

No te preocupes si no ves cómo funciona este código de inmediato. Vamos a trabajar juntos, ya que es un gran ejercicio de pensamiento recursivo. La función interna `find` es la que realiza la recursión real. Toma dos argumentos: el número actual y una cadena que registra cómo llegamos a este número. Si encuentra una solución, devuelve una cadena que muestra cómo llegar al objetivo. Si no puede encontrar una solución comenzando desde este número, devuelve `null`.

Para hacer esto, la función realiza una de tres acciones. Si el número actual es el número objetivo, el historial actual es una forma de alcanzar ese objetivo, por lo que se devuelve. Si el número actual es mayor que el objetivo, no tiene sentido explorar más esta rama porque tanto la suma como la multiplicación solo harán que el número sea más grande, por lo que devuelve `null`. Finalmente, si aún estamos por debajo del número objetivo, la función prueba ambas rutas posibles que parten del número actual llamándose a sí

misma dos veces, una vez para la suma y otra vez para la multiplicación. Si la primera llamada devuelve algo que no es `null`, se devuelve. De lo contrario, se devuelve la segunda llamada, independientemente de si produce una cadena o `null`.

Para entender mejor cómo esta función produce el efecto que estamos buscando, veamos todas las llamadas a `find` que se hacen al buscar una solución para el número 13:

```
find(1, "1")
  find(6, "(1 + 5)")
    find(11, "((1 + 5) + 5)")
      find(16, "(((1 + 5) + 5) + 5)")
        demasiado grande
      find(33, "(((1 + 5) + 5) * 3)")
        demasiado grande
    find(18, "((1 + 5) * 3)")
      demasiado grande
  find(3, "(1 * 3)")
    find(8, "((1 * 3) + 5)")
      find(13, "(((1 * 3) + 5) + 5)")
        ¡encontrado!
```

La sangría indica la profundidad de la pila de llamadas. La primera vez que se llama a `find`, la función comienza llamándose a sí misma para explorar la solución que comienza con $(1 + 5)$. Esa llamada seguirá recursivamente para explorar *cada* solución continua que produzca un número menor o igual al número objetivo. Como no encuentra uno que alcance el objetivo, devuelve `null` a la primera llamada. Allí, el operador `??` hace que ocurra la llamada que explora $(1 * 3)$. Esta búsqueda tiene más suerte: su primera llamada recursiva, a través de otra llamada recursiva, alcanza el número objetivo. Esa llamada más interna devuelve una cadena, y cada uno

de los operadores ?? en las llamadas intermedias pasa esa cadena, devolviendo en última instancia la solución.

CRECIMIENTO DE FUNCIONES

Hay dos formas más o menos naturales de introducir funciones en los programas.

La primera ocurre cuando te encuentras escribiendo código similar varias veces. Preferirías no hacer eso, ya que tener más código significa más espacio para que se escondan los errores y más material para que las personas que intentan entender el programa lo lean. Por lo tanto, tomas la funcionalidad repetida, encuentras un buen nombre para ella y la colocas en una función.

La segunda forma es que te das cuenta de que necesitas alguna funcionalidad que aún no has escrito y que suena como si mereciera su propia función. Comienzas por nombrar la función, luego escribes su cuerpo. Incluso podrías comenzar a escribir código que use la función antes de definir la función en sí.

Lo difícil que es encontrar un buen nombre para una función es una buena indicación de lo claro que es el concepto que estás tratando de envolver. Vamos a través de un ejemplo.

Queremos escribir un programa que imprima dos números: el número de vacas y de pollos en una granja, con las palabras `Vacas` y `Pollos` después de ellos y ceros rellenos antes de ambos números para que siempre tengan tres dígitos:

```
007 Vacas  
011 Pollos
```

Esto pide una función con dos argumentos: el número de vacas y el número de pollos. ¡Vamos a programar!

```
function imprimirInventarioGranja(vacas, pollos) {  
  let cadenaVaca = String(vacas);  
  while (cadenaVaca.length < 3) {  
    cadenaVaca = "0" + cadenaVaca;  
  }  
  console.log(`${cadenaVaca} Vacas`);  
  let cadenaPollo = String(pollos);  
  while (cadenaPollo.length < 3) {  
    cadenaPollo = "0" + cadenaPollo;  
  }  
  console.log(`${cadenaPollo} Pollos`);  
}  
imprimirInventarioGranja(7, 11);
```

Escribir `.length` después de una expresión de cadena nos dará la longitud de esa cadena. Por lo tanto, los bucles `while` e `if` siguen añadiendo ceros delante de las cadenas de números hasta que tengan al menos tres caracteres de longitud.

¡Misión cumplida! Pero justo cuando estamos a punto de enviarle al granjero el código (junto con una jugosa factura), ella llama y nos dice que también ha comenzado a criar cerdos, ¿podríamos extender el software para imprimir también los cerdos?

¡Claro que podemos! Pero justo cuando estamos en el proceso de copiar y pegar esas cuatro líneas una vez más, nos detenemos y reconsideramos. Tiene que haber una mejor manera. Aquí está un primer intento:

```
function imprimirConRellenoYEtiqueta(numero, etiqueta) {  
  let cadenaNumero = String(numero);  
  while (cadenaNumero.length < 3) {
```

```

        cadenaNumero = "0" + cadenaNumero;
    }
    console.log(`${cadenaNumero} ${etiqueta}`);
}

function imprimirInventarioGranja(vacas, pollos, cerdos) {
    imprimirConRellenoYEtiqueta(vacas, "Vacas");
    imprimirConRellenoYEtiqueta(pollos, "Pollos");
    imprimirConRellenoYEtiqueta(cerdos, "Cerdos");
}

imprimirInventarioGranja(7, 11, 3);

```

¡Funciona! Pero ese nombre, `imprimirConRellenoYEtiqueta`, es un poco incómodo. Confluye tres cosas: imprimir, rellenar con ceros y añadir una etiqueta, en una sola función.

En lugar de sacar la parte repetida de nuestro programa completamente, intentemos sacar un solo *concepto*:

```

function rellenarConCeros(numero, ancho) {
    let cadena = String(numero);
    while (cadena.length < ancho) {
        cadena = "0" + cadena;
    }
    return cadena;
}

function imprimirInventarioGranja(vacas, pollos, cerdos) {
    console.log(`${rellenarConCeros(vacas, 3)} Vacas`);
    console.log(`${rellenarConCeros(pollos, 3)} Pollos`);
    console.log(`${rellenarConCeros(cerdos, 3)} Cerdos`);
}

imprimirInventarioGranja(7, 16, 3);

```

Una función con un nombre claro y obvio como `rellenarConCeros` hace que sea más fácil para alguien que lee el

código entender qué hace. Además, una función así es útil en más situaciones que solo este programa específico. Por ejemplo, podrías usarla para ayudar a imprimir tablas de números alineadas correctamente.

¿Qué tan inteligente y versátil *debería* ser nuestra función? Podríamos escribir cualquier cosa, desde una función terriblemente simple que solo puede rellenar un número para que tenga tres caracteres de ancho hasta un sistema de formato de números generalizado complicado que maneje números fraccionarios, números negativos, alineación de puntos decimales, relleno con diferentes caracteres, y más.

Un principio útil es abstenerse de agregar ingenio a menos que estés absolutamente seguro de que lo vas a necesitar. Puede ser tentador escribir “marcos de trabajo” generales para cada trozo de funcionalidad que te encuentres. Resiste esa tentación. No lograrás hacer ningún trabajo real: estarás demasiado ocupado escribiendo código que nunca usas.

FUNCIONES Y EFECTOS SECUNDARIOS

Las funciones pueden dividirse aproximadamente en aquellas que se llaman por sus efectos secundarios y aquellas que se llaman por su valor de retorno (aunque también es posible tener efectos secundarios y devolver un valor).

La primera función auxiliar en el ejemplo de la granja, `imprimirRellenadoConEtiqueta`, se llama por su efecto secundario: imprime una línea. La segunda versión, `rellenarConCero`, se llama por su valor de retorno. No es

casualidad que la segunda sea útil en más situaciones que la primera. Las funciones que crean valores son más fáciles de combinar de nuevas formas que las funciones que realizan efectos secundarios directamente.

Una función *pura* es un tipo específico de función productora de valor que no solo no tiene efectos secundarios, sino que tampoco depende de efectos secundarios de otro código, por ejemplo, no lee enlaces globales cuyo valor podría cambiar. Una función pura tiene la agradable propiedad de que, al llamarla con los mismos argumentos, siempre produce el mismo valor (y no hace nada más). Una llamada a tal función puede sustituirse por su valor de retorno sin cambiar el significado del código. Cuando no estás seguro de que una función pura esté funcionando correctamente, puedes probarla llamándola y saber que si funciona en ese contexto, funcionará en cualquier otro. Las funciones no puras tienden a requerir más andamiaje para probarlas.

Aún así, no hay necesidad de sentirse mal al escribir funciones que no son puras. Los efectos secundarios a menudo son útiles. No hay forma de escribir una versión pura de `console.log`, por ejemplo, y es bueno tener `console.log`. Algunas operaciones también son más fáciles de expresar de manera eficiente cuando usamos efectos secundarios.

RESUMEN

Este capítulo te enseñó cómo escribir tus propias funciones. La palabra clave `function`, cuando se usa como expresión, puede crear un valor de función. Cuando se usa como una declaración,

puede usarse para declarar un enlace y darle una función como su valor. Las funciones de flecha son otra forma de crear funciones.

```
// Definir f para contener un valor de función
const f = function(a) {
  console.log(a + 2);
};

// Declarar g como una función
function g(a, b) {
  return a * b * 3.5;
}

// Un valor de función menos verboso
let h = a => a % 3;
```

Una parte clave para entender las funciones es comprender los ámbitos (scopes). Cada bloque crea un nuevo ámbito. Los parámetros y las vinculaciones declaradas en un ámbito dado son locales y no son visibles desde el exterior. Las vinculaciones declaradas con `var` se comportan de manera diferente: terminan en el ámbito de la función más cercana o en el ámbito global.

Separar las tareas que realiza tu programa en diferentes funciones es útil. No tendrás que repetirte tanto, y las funciones pueden ayudar a organizar un programa agrupando el código en piezas que hacen cosas específicas.

EJERCICIOS

MÍNIMO

El [capítulo previo](#) presentó la función estándar `Math.min` que devuelve su menor argumento. Ahora podemos escribir una función

como esa nosotros mismos. Define la función `min` que toma dos argumentos y devuelve su mínimo.

RECURSIÓN

Hemos visto que podemos usar `%` (el operador de resto) para verificar si un número es par o impar al usar `% 2` para ver si es divisible por dos. Aquí hay otra forma de definir si un número entero positivo es par o impar:

- El cero es par.
- El uno es impar.
- Para cualquier otro número N , su paridad es la misma que $N - 2$.

Define una función recursiva `isEven` que corresponda a esta descripción. La función debe aceptar un solo parámetro (un número entero positivo) y devolver un booleano.

Pruébalo con 50 y 75. Observa cómo se comporta con -1. ¿Por qué? ¿Puedes pensar en una forma de solucionarlo?

CONTANDO FRIJOLES

Puedes obtener el n -ésimo carácter, o letra, de una cadena escribiendo `[N]` después de la cadena (por ejemplo, `cadena[2]`). El valor resultante será una cadena que contiene solo un carácter (por ejemplo, `"b"`). El primer carácter tiene la posición 0, lo que hace que el último se encuentre en la posición `cadena.length - 1`. En otras palabras, una cadena de dos caracteres tiene longitud 2, y sus caracteres tienen posiciones 0 y 1.

Escribe una función `contarBs` que tome una cadena como único argumento y devuelva un número que indique cuántos caracteres B en mayúscula hay en la cadena.

A continuación, escribe una función llamada `contarCaracter` que se comporte como `contarBs`, excepto que toma un segundo argumento que indica el carácter que se va a contar (en lugar de contar solo caracteres B en mayúscula). Reescribe `contarBs` para hacer uso de esta nueva función.

ESTRUCTURAS DE DATOS: OBJETOS Y ARRAYS

“En dos ocasiones me han preguntado: ‘Dígame, Sr. Babbage, si introduce en la máquina cifras erróneas, ¿saldrán respuestas correctas?’ [...] No soy capaz de entender correctamente el tipo de confusión de ideas que podría provocar tal pregunta.”

—Charles Babbage, *Passages from the Life of a Philosopher* (1864)



Números, booleanos y cadenas de texto son los átomos a partir de los cuales se construyen las estructuras de datos. Sin embargo, muchos tipos de información requieren más de un átomo. Los *objetos* nos permiten agrupar valores, incluyendo otros objetos, para construir estructuras más complejas.

Hasta ahora, los programas que hemos creado han estado limitados por el hecho de que operaban solo en tipos de datos simples.

Después de aprender los conceptos básicos de estructuras de datos en este capítulo, sabrás lo suficiente como para comenzar a escribir programas útiles.

El capítulo trabajará a través de un ejemplo de programación más o menos realista, introduciendo conceptos a medida que se aplican al problema en cuestión. El código de ejemplo a menudo se basará en funciones y variables introducidas anteriormente en el libro.

EL HOMBREARDILLA

De vez en cuando, usualmente entre las 8 p. m. y las 10 p. m., Jacques se encuentra transformándose en un pequeño roedor peludo con una cola espesa.

Por un lado, Jacques está bastante contento de no tener licantrópía clásica. Convertirse en una ardilla causa menos problemas que convertirse en un lobo. En lugar de preocuparse por comer accidentalmente al vecino (*eso sería incómodo*), se preocupa por ser comido por el gato del vecino. Después de dos ocasiones de despertar en una rama precariamente delgada en la copa de un roble, desnudo y desorientado, ha optado por cerrar con llave las puertas y ventanas de su habitación por la noche y poner unas cuantas nueces en el suelo para mantenerse ocupado.

Pero Jacques preferiría deshacerse por completo de su condición. Las ocurrencias irregulares de la transformación hacen que sospeche que podrían ser desencadenadas por algo. Durante un tiempo, creyó

que sucedía solo en días en los que había estado cerca de robles. Sin embargo, evitar los robles no resolvió el problema.

Cambió a un enfoque más científico, Jacques ha comenzado a llevar un registro diario de todo lo que hace en un día dado y si cambió de forma. Con estos datos, espera estrechar las condiciones que desencadenan las transformaciones. Lo primero que necesita es una estructura de datos para almacenar esta información.

CONJUNTOS DE DATOS

Para trabajar con un conjunto de datos digitales, primero tenemos que encontrar una forma de representarlo en la memoria de nuestra máquina. Digamos, por ejemplo, que queremos representar una colección de los números 2, 3, 5, 7 y 11.

Podríamos ser creativos con las cadenas, después de todo, las cadenas pueden tener cualquier longitud, por lo que podemos poner muchos datos en ellas, y usar "2 3 5 7 11" como nuestra representación. Pero esto es incómodo. Tendríamos que extraer de alguna manera los dígitos y convertirlos de vuelta a números para acceder a ellos.

Afortunadamente, JavaScript proporciona un tipo de dato específicamente para almacenar secuencias de valores. Se llama un *array* y se escribe como una lista de valores entre corchetes, separados por comas:

```
let listaDeNumeros = [2, 3, 5, 7, 11];
console.log(listaDeNumeros[2]);
// → 5
console.log(listaDeNumeros[0]);
```

```
// → 2  
console.log(listaDeNumeros[2 - 1]);  
// → 3
```

La notación para acceder a los elementos dentro de un array también utiliza corchetes. Un par de corchetes inmediatamente después de una expresión, con otra expresión dentro de ellos, buscará el elemento en la expresión de la izquierda que corresponde al *índice* dado por la expresión en los corchetes.

El primer índice de un array es cero, no uno, por lo que el primer elemento se recupera con `listaDeNumeros[0]`. El conteo basado en cero tiene una larga tradición en tecnología y de ciertas maneras tiene mucho sentido, pero requiere cierta acostumbrarse. Piensa en el índice como el número de elementos a omitir, contando desde el inicio del array.

PROPIEDADES

Hemos visto algunas expresiones como `miCadena.length` (para obtener la longitud de una cadena) y `Math.max` (la función máxima) en capítulos anteriores. Estas expresiones acceden a una *propiedad* de algún valor. En el primer caso, accedemos a la propiedad `length` del valor en `miCadena`. En el segundo, accedemos a la propiedad llamada `max` en el objeto `Math` (que es una colección de constantes y funciones relacionadas con matemáticas).

Casi todos los valores de JavaScript tienen propiedades. Las excepciones son `null` y `undefined`. Si intentas acceder a una propiedad en uno de estos valores no definidos, obtendrás un error:

```
null.length;  
// → TypeError: null no tiene propiedades
```

Las dos formas principales de acceder a propiedades en JavaScript son con un punto y con corchetes. Tanto `valor.x` como `valor[x]` acceden a una propiedad en `valor`, pero no necesariamente a la misma propiedad. La diferencia radica en cómo se interpreta `x`. Al usar un punto, la palabra después del punto es el nombre literal de la propiedad. Al usar corchetes, la expresión entre los corchetes es *evaluada* para obtener el nombre de la propiedad. Mientras que `valor.x` obtiene la propiedad de `valor` llamada “x”, `valor[x]` toma el valor de la variable llamada `x` y lo utiliza, convertido a cadena, como nombre de propiedad. Si sabes que la propiedad en la que estás interesado se llama *color*, dices `valor.color`. Si quieres extraer la propiedad nombrada por el valor almacenado en la vinculación `i`, dices `valor[i]`. Los nombres de las propiedades son cadenas de texto. Pueden ser cualquier cadena, pero la notación de punto solo funciona con nombres que parecen nombres de vinculaciones válidos, comenzando con una letra o guion bajo, y conteniendo solo letras, números y guiones bajos. Si deseas acceder a una propiedad llamada `2` o *John Doe*, debes utilizar corchetes: `valor[2]` o `valor["John Doe"]`.

Los elementos en un array se almacenan como propiedades del array, utilizando números como nombres de propiedades. Dado que no puedes usar la notación de punto con números y generalmente quieres usar una vinculación que contenga el índice de todos modos, debes utilizar la notación de corchetes para acceder a ellos.

Al igual que las cadenas de texto, los arrays tienen una propiedad `length` que nos dice cuántos elementos tiene el array.

MÉTODOS

Tanto los valores de cadena como los de array contienen, además de la propiedad `length`, varias propiedades que contienen valores de función.

```
let doh = "Doh";  
console.log(typeof doh.toUpperCase);  
// → función  
console.log(doh.toUpperCase());  
// → DOH
```

Cada cadena de texto tiene una propiedad `toUpperCase`. Cuando se llama, devolverá una copia de la cadena en la que todas las letras se han convertido a mayúsculas. También existe `toLowerCase`, que hace lo contrario.

Curiosamente, aunque la llamada a `toUpperCase` no pasa argumentos, de alguna manera la función tiene acceso a la cadena "Doh", el valor cuya propiedad llamamos. Descubrirás cómo funciona esto en [Capítulo 6](#).

Las propiedades que contienen funciones generalmente se llaman *métodos* del valor al que pertenecen, como en “`toUpperCase` es un método de una cadena”.

Este ejemplo demuestra dos métodos que puedes utilizar para manipular arrays:

```
let secuencia = [1, 2, 3];
secuencia.push(4);
secuencia.push(5);
console.log(secuencia);
// → [1, 2, 3, 4, 5]
console.log(secuencia.pop());
// → 5
console.log(secuencia);
// → [1, 2, 3, 4]
```

El método `push` agrega valores al final de un array. El método `pop` hace lo opuesto, eliminando el último valor en el array y devolviéndolo.

Estos nombres un tanto tontos son términos tradicionales para operaciones en una *pila*. Una pila, en programación, es una estructura de datos que te permite agregar valores a ella y sacarlos en el orden opuesto para que lo que se agregó último se elimine primero. Las pilas son comunes en programación; es posible que recuerdes la función `call stack` del [capítulo anterior](#), que es una instancia de la misma idea.

OBJETOS

De vuelta al hombre-ardilla. Un conjunto de entradas de registro diario se puede representar como un array, pero las entradas no consisten solo en un número o una cadena, cada entrada necesita almacenar una lista de actividades y un valor booleano que indique si Jacques se convirtió en ardilla o no. Idealmente, nos gustaría agrupar estos elementos en un único valor y luego poner esos valores agrupados en un array de entradas de registro.

Los valores del tipo `object` son colecciones arbitrarias de propiedades. Una forma de crear un objeto es usando llaves como una expresión:

```
let dia1 = {
  hombreArdilla: false,
  eventos: ["trabajo", "tocó árbol", "pizza", "correr"]
};
console.log(dia1.hombreArdilla);
// → false
console.log(dia1.lobo);
// → undefined
dia1.lobo = false;
console.log(dia1.lobo);
// → false
```

Dentro de las llaves, se escribe una lista de propiedades separadas por comas. Cada propiedad tiene un nombre seguido por dos puntos y un valor. Cuando un objeto se escribe en varias líneas, indentarlo como se muestra en este ejemplo ayuda a la legibilidad. Las propiedades cuyos nombres no son nombres de enlace válidos o números válidos deben ir entre comillas:

```
let descripciones = {
  trabajo: "Fui a trabajar",
  "tocó árbol": "Tocó un árbol"
};
```

Esto significa que las llaves tienen *dos* significados en JavaScript. Al principio de una sentencia, comienzan un bloque de sentencias. En cualquier otra posición, describen un objeto. Afortunadamente, rara vez es útil comenzar una sentencia con un objeto entre llaves, por lo que la ambigüedad entre estos dos casos no es gran problema. El único caso en el que esto surge es cuando quiere devolver un objeto

desde una función flecha abreviada: no puede escribir `n => {prop: n}`, ya que las llaves se interpretarán como el cuerpo de una función. En cambio, debe poner un conjunto de paréntesis alrededor del objeto para dejar claro que es una expresión.

Al leer una propiedad que no existe, obtendrás el valor `undefined`.

Es posible asignar un valor a una expresión de propiedad con el operador `=`. Esto reemplazará el valor de la propiedad si ya existía o creará una nueva propiedad en el objeto si no existía.

Para volver brevemente a nuestro modelo de tentáculos de enlaces, los enlaces de propiedad son similares. *Agarran* valores, pero otros enlaces y propiedades podrían estar aferrándose a esos mismos valores. Puedes pensar en los objetos como pulpos con cualquier cantidad de tentáculos, cada uno con un nombre escrito en él.

El operador `delete` corta un tentáculo de dicho pulpo. Es un operador unario que, cuando se aplica a una propiedad de un objeto, eliminará la propiedad nombrada del objeto. Esto no es algo común de hacer, pero es posible.

```
let unObjeto = {izquierda: 1, derecha: 2};
console.log(unObjeto.izquierda);
// → 1
delete unObjeto.izquierda;
console.log(unObjeto.izquierda);
// → undefined
console.log("izquierda" in unObjeto);
// → false
console.log("derecha" in unObjeto);
// → true
```

El operador binario `in`, cuando se aplica a una cadena y un objeto, te dice si ese objeto tiene una propiedad con ese nombre. La diferencia entre establecer una propiedad como `undefined` y realmente borrarla es que, en el primer caso, el objeto todavía *tiene* la propiedad (simplemente no tiene un valor muy interesante), mientras que en el segundo caso la propiedad ya no está presente y `in` devolverá `false`.

Para averiguar qué propiedades tiene un objeto, puedes utilizar la función `Object.keys`. Al darle la función un objeto, devolverá un array de cadenas: los nombres de las propiedades del objeto:

```
console.log(Object.keys({x: 0, y: 0, z: 2}));  
// → ["x", "y", "z"]
```

Existe una función `Object.assign` que copia todas las propiedades de un objeto en otro:

```
let objetoA = {a: 1, b: 2};  
Object.assign(objetoA, {b: 3, c: 4});  
console.log(objetoA);  
// → {a: 1, b: 3, c: 4}
```

Los arrays, entonces, son solo un tipo de objeto especializado para almacenar secuencias de cosas. Si evalúas `typeof []`, producirá `"object"`. Puedes visualizar los arrays como pulpos largos y planos con todos sus tentáculos en una fila ordenada, etiquetados con números.

Jacques representará el diario que lleva como un array de objetos:

```
let diario = [  
  {eventos: ["trabajo", "tocó árbol", "pizza",
```

```

        "corrió", "televisión"],
    ardilla: false},
    {eventos: ["trabajo", "helado", "coliflor",
        "lasaña", "tocó árbol", "se cepilló los dientes"],
    ardilla: false},
    {eventos: ["fin de semana", "ciclismo", "descanso",
        "cacahuetes",
        "cerveza"],
    ardilla: true},
    /* y así sucesivamente... */
];

```

MUTABILIDAD

Pronto llegaremos a la programación real, pero primero, hay una pieza más de teoría para entender.

Vimos que los valores de objetos pueden modificarse. Los tipos de valores discutidos en capítulos anteriores, como números, cadenas y booleanos, son todos *inmutables*—es imposible cambiar valores de esos tipos. Puedes combinarlos y derivar nuevos valores de ellos, pero al tomar un valor específico de cadena, ese valor siempre permanecerá igual. El texto dentro de él no puede ser cambiado. Si tienes una cadena que contiene "gato", no es posible que otro código cambie un carácter en tu cadena para que diga "rata".

Los objetos funcionan de manera diferente. *Puedes* cambiar sus propiedades, lo que hace que un valor de objeto tenga un contenido diferente en momentos diferentes.

Cuando tenemos dos números, 120 y 120, podemos considerarlos precisamente el mismo número, tanto si se refieren a los mismos bits físicos como si no. Con los objetos, hay una diferencia entre tener dos

referencias al mismo objeto y tener dos objetos diferentes que contienen las mismas propiedades. Considera el siguiente código:

```
let object1 = {value: 10};
let object2 = object1;
let object3 = {value: 10};

console.log(object1 == object2);
// → true
console.log(object1 == object3);
// → false

object1.value = 15;
console.log(object2.value);
// → 15
console.log(object3.value);
// → 10
```

Las asignaciones `object1` y `object2` contienen la *misma* referencia al objeto, por lo que al cambiar `object1` también se cambia el valor de `object2`. Se dice que tienen la misma *identidad*. La asignación `object3` apunta a un objeto diferente, que inicialmente contiene las mismas propiedades que `object1` pero vive una vida separada.

Las asignaciones pueden ser modificables o constantes, pero esto es independiente de cómo se comportan sus valores. Aunque los valores numéricos no cambian, puedes utilizar una asignación `let` para hacer un seguimiento de un número que cambia al cambiar el valor al que apunta la asignación. De manera similar, aunque una asignación `const` a un objeto en sí no puede cambiarse y seguirá apuntando al mismo objeto, los *contenidos* de ese objeto pueden cambiar.

```
const score = {visitors: 0, home: 0};  
// Esto está bien  
score.visitors = 1;  
// Esto no está permitido  
score = {visitors: 1, home: 1};
```

Cuando se comparan objetos con el operador `==` de JavaScript, se compara por identidad: producirá `true` solo si ambos objetos son exactamente el mismo valor. Comparar objetos diferentes devolverá `false`, incluso si tienen propiedades idénticas. No hay una operación de comparación “profunda” incorporada en JavaScript que compare objetos por contenido, pero es posible escribirla tú mismo (lo cual es uno de los [ejercicios](#) al final de este capítulo).

EL DIARIO DEL LICÁNTROPO

Jacques inicia su intérprete de JavaScript y configura el entorno que necesita para mantener su diario:

```
let journal = [];  
  
function addEntry(events, squirrel) {  
  journal.push({events, squirrel});  
}
```

Observa que el objeto agregado al diario luce un poco extraño. En lugar de declarar propiedades como `events: events`, simplemente se da un nombre de propiedad: `events`. Esta es una forma abreviada que significa lo mismo: si un nombre de propiedad en notación de llaves no va seguido de un valor, su valor se toma del enlace con el mismo nombre.

Cada noche a las 10 p.m., o a veces a la mañana siguiente después de bajar de la repisa superior de su estantería, Jacques registra el día:




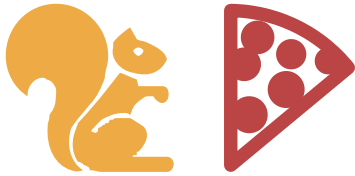
```
addEntry(["work", "touched tree", "pizza", "running",  
         "television"], false);  
addEntry(["work", "ice cream", "cauliflower", "lasagna",  
         "touched tree", "brushed teeth"], false);  
addEntry(["weekend", "cycling", "break", "peanuts",  
         "beer"], true);
```

Una vez que tiene suficientes puntos de datos, tiene la intención de utilizar estadísticas para descubrir qué eventos pueden estar relacionados con las transformaciones en ardilla.

La *correlación* es una medida de la dependencia entre variables estadísticas. Una variable estadística no es exactamente igual a una variable de programación. En estadística, típicamente tienes un conjunto de *mediciones*, y cada variable se mide para cada medición. La correlación entre variables suele expresarse como un valor que va de -1 a 1. Una correlación de cero significa que las variables no están relacionadas. Una correlación de 1 indica que las dos están perfectamente relacionadas: si conoces una, también conoces la otra. Un -1 también significa que las variables están perfectamente relacionadas pero son opuestas: cuando una es verdadera, la otra es falsa.

Para calcular la medida de correlación entre dos variables booleanas, podemos utilizar el *coeficiente phi* (ϕ). Esta es una fórmula cuya entrada es una tabla de frecuencias que contiene la cantidad de veces que se observaron las diferentes combinaciones de las variables. La salida de la fórmula es un número entre -1 y 1 que describe la correlación.

Podríamos tomar el evento de comer pizza y ponerlo en una tabla de frecuencias como esta, donde cada número indica la cantidad de veces que ocurrió esa combinación en nuestras mediciones.

 No squirrel, no pizza 76	 No squirrel, pizza 9
 Squirrel, no pizza 4	 Squirrel, pizza 1

Si llamamos a esa tabla n , podemos calcular ϕ utilizando la siguiente fórmula:

$$\phi = \frac{n_{11}n_{00} - n_{10}n_{01}}{\sqrt{n_{1.}n_{0.}n_{.1}n_{.0}}}$$

(Si en este punto estás dejando el libro para concentrarte en un terrible flashback a la clase de matemáticas de décimo grado, ¡espera! No pretendo torturarte con interminables páginas de notación críptica, solo es esta fórmula por ahora. Y incluso con esta, todo lo que haremos es convertirla en JavaScript).

La notación n_{01} indica la cantidad de mediciones donde la primera variable (ardillez) es falsa (0) y la segunda variable (pizza) es verdadera (1). En la tabla de pizza, n_{01} es 9. El valor $n_{1.}$ se refiere a la

suma de todas las mediciones donde la primera variable es verdadera, que es 5 en el ejemplo de la tabla. De manera similar, n_{\circ} se refiere a la suma de las mediciones donde la segunda variable es falsa.

Entonces para la tabla de pizza, la parte encima de la línea de división (el dividendo) sería $1 \times 76 - 4 \times 9 = 40$, y la parte debajo de ella (el divisor) sería la raíz cuadrada de $5 \times 85 \times 10 \times 80$, o $\sqrt{340,000}$. Esto da un valor de $\phi \approx 0.069$, que es muy pequeño. Comer pizza no parece tener influencia en las transformaciones.

CALCULANDO LA CORRELACIÓN

Podemos representar una tabla dos por dos en JavaScript con un array de cuatro elementos (`[76, 9, 4, 1]`). También podríamos usar otras representaciones, como un array que contiene dos arrays de dos elementos cada uno (`[[76, 9], [4, 1]]`) o un objeto con nombres de propiedades como `"11"` y `"01"`, pero el array plano es simple y hace que las expresiones que acceden a la tabla sean agradablemente cortas. Interpretaremos los índices del array como números binarios de dos bits, donde el dígito más a la izquierda (más significativo) se refiere a la variable ardilla y el dígito más a la derecha (menos significativo) se refiere a la variable de evento. Por ejemplo, el número binario `10` se refiere al caso donde Jacques se transformó en ardilla, pero el evento (digamos, "pizza") no ocurrió. Esto sucedió cuatro veces. Y como `10` en binario es `2` en notación decimal, almacenaremos este número en el índice `2` del array.

Esta es la función que calcula el coeficiente ϕ a partir de dicho array:


```
function phi(table) {
  return (table[3] * table[0] - table[2] * table[1]) /
    Math.sqrt((table[2] + table[3]) *
      (table[0] + table[1]) *
      (table[1] + table[3]) *
      (table[0] + table[2]));
}

console.log(phi([76, 9, 4, 1]));
// → 0.068599434
```

Esta es una traducción directa de la fórmula de ϕ a JavaScript. `Math.sqrt` es la función de raíz cuadrada, como se provee en el objeto `Math` en un entorno estándar de JavaScript. Debemos agregar dos campos de la tabla para obtener campos como n_i , porque las sumas de filas o columnas no se almacenan directamente en nuestra estructura de datos.

Jacques mantiene su diario por tres meses. El conjunto de datos resultante está disponible en el [sandbox de código](https://eloquentjavascript.net/code#4) para este capítulo (<https://eloquentjavascript.net/code#4>), donde se almacena en el vínculo JOURNAL, y en un archivo descargable [aquí](#).

Para extraer una tabla dos por dos para un evento específico del diario, debemos recorrer todas las entradas y contar cuántas veces ocurre el evento en relación con las transformaciones de ardilla:

```
function tableFor(event, journal) {
  let table = [0, 0, 0, 0];
  for (let i = 0; i < journal.length; i++) {
    let entry = journal[i], index = 0;
    if (entry.events.includes(event)) index += 1;
    if (entry.squirrel) index += 2;
    table[index] += 1;
  }
  return table;
}
```

```
}  
  
console.log(tableFor("pizza", JOURNAL));  
// → [76, 9, 4, 1]
```

Los arrays tienen un método `includes` que comprueba si un valor dado existe en el array. La función utiliza esto para determinar si el nombre del evento en el que está interesado forma parte de la lista de eventos de un día dado.

El cuerpo del bucle en `tableFor` determina en qué caja de la tabla cae cada entrada del diario, verificando si la entrada contiene el evento específico en el que está interesado y si el evento ocurre junto con un incidente de ardilla. Luego, el bucle suma uno a la caja correcta de la tabla.

Ahora tenemos las herramientas necesarias para calcular correlaciones individuales. El único paso restante es encontrar una correlación para cada tipo de evento que se registró y ver si algo destaca.

BUCLES DE ARRAY

En la función `tableFor`, hay un bucle como este:

```
for (let i = 0; i < JOURNAL.length; i++) {  
  let entry = JOURNAL[i];  
  // Hacer algo con entry  
}
```

Este tipo de bucle es común en el JavaScript clásico; recorrer arrays elemento por elemento es algo que se hace con frecuencia, y para

hacerlo se recorre un contador sobre la longitud del array y se selecciona cada elemento por turno.

Hay una forma más sencilla de escribir tales bucles en JavaScript moderno:

```
for (let entry of JOURNAL) {  
  console.log(`${entry.events.length} eventos.`);  
}
```

Cuando un bucle `for` usa la palabra `of` después de la definición de su variable, recorrerá los elementos del valor dado después de `of`. Esto no solo funciona para arrays, sino también para cadenas y algunas otras estructuras de datos. Discutiremos *cómo* funciona en [Capítulo 6](#).

EL ANÁLISIS FINAL

Necesitamos calcular una correlación para cada tipo de evento que ocurre en el conjunto de datos. Para hacerlo, primero necesitamos *encontrar* cada tipo de evento.

```
function journalEvents(journal) {  
  let events = [];  
  for (let entry of journal) {  
    for (let event of entry.events) {  
      if (!events.includes(event)) {  
        events.push(event);  
      }  
    }  
  }  
  return events;  
}  
  
console.log(journalEvents(JOURNAL));  
// → ["zanahoria", "ejercicio", "fin de semana", "pan", ...]
```

Agregando los nombres de cualquier evento que no estén en él al array `events`, la función recopila todos los tipos de eventos.

Usando esa función, podemos ver todas las correlaciones:

```
for (let event of journalEvents(JOURNAL)) {  
  console.log(event + ":", phi(tableFor(event, JOURNAL)));  
}  
// → zanahoria:    0.0140970969  
// → ejercicio: 0.0685994341  
// → fin de semana: 0.1371988681  
// → pan:    -0.0757554019  
// → pudín: -0.0648203724  
// y así sucesivamente...
```

La mayoría de las correlaciones parecen estar cerca de cero. Comer zanahorias, pan o pudín aparentemente no desencadena la licantrópía de las ardillas. Las transformaciones parecen ocurrir un poco más a menudo los fines de semana. Filtraremos los resultados para mostrar solo correlaciones mayores que 0.1 o menores que -0.1:

```
for (let event of journalEvents(JOURNAL)) {  
  let correlation = phi(tableFor(event, JOURNAL));  
  if (correlation > 0.1 || correlation < -0.1) {  
    console.log(event + ":", correlation);  
  }  
}  
// → fin de semana:    0.1371988681  
// → cepillarse los dientes: -0.3805211953  
// → dulces:    0.1296407447  
// → trabajo:    -0.1371988681  
// → espaguetis:    0.2425356250  
// → lectura:    0.1106828054  
// → cacahuetes:    0.5902679812
```

¡Ajá! Hay dos factores con una correlación claramente más fuerte que los demás. Comer cacahuets tiene un fuerte efecto positivo en la posibilidad de convertirse en una ardilla, mientras que cepillarse los dientes tiene un efecto negativo significativo.

Interesante. Intentemos algo:

```
for (let entry of JOURNAL) {  
  if (entry.events.includes("cacahuets") &&  
      !entry.events.includes("cepillarse los dientes")) {  
    entry.events.push("dientes de cacahuete");  
  }  
}  
console.log(phi(tableFor("dientes de cacahuete", JOURNAL)));  
// → 1
```

Ese es un resultado sólido. El fenómeno ocurre precisamente cuando Jacques come cacahuets y no se cepilla los dientes. Si tan solo no fuera tan descuidado con la higiene dental, ni siquiera se habría dado cuenta de su aflicción.

Sabiendo esto, Jacques deja de comer cacahuets por completo y descubre que sus transformaciones se detienen.

Pero solo pasan unos pocos meses antes de que se dé cuenta de que algo falta en esta forma de vivir completamente humana. Sin sus aventuras salvajes, Jacques apenas se siente vivo. Decide que prefiere ser un animal salvaje a tiempo completo. Después de construir una hermosa casita en un árbol en el bosque y equiparla con un dispensador de mantequilla de cacahuete y un suministro de diez años de mantequilla de cacahuete, cambia de forma por última vez y vive la corta y enérgica vida de una ardilla.

MÁS ARREOLOGÍA

Antes de terminar el capítulo, quiero presentarte algunos conceptos más relacionados con objetos. Comenzaré presentando algunos métodos de array generalmente útiles.

Vimos push y pop, que agregan y eliminan elementos al final de un array, [anteriormente](#) en este capítulo. Los métodos correspondientes para agregar y eliminar cosas al principio de un array se llaman unshift y shift.

```
let listaDeTareas = [];  
function recordar(tarea) {  
  listaDeTareas.push(tarea);  
}  
function obtenerTarea() {  
  return listaDeTareas.shift();  
}  
function recordarUrgente(tarea) {  
  listaDeTareas.unshift(tarea);  
}
```

Este programa gestiona una cola de tareas. Agregas tareas al final de la cola llamando a `recordar("comestibles")`, y cuando estás listo para hacer algo, llamas a `obtenerTarea()` para obtener (y eliminar) el primer elemento de la cola. La función `recordarUrgente` también agrega una tarea pero la agrega al principio en lugar de al final de la cola.

Para buscar un valor específico, los arrays proporcionan un método `indexOf`. Este método busca a través del array desde el principio hasta el final y devuelve el índice en el que se encontró el valor solicitado, o -1 si no se encontró. Para buscar desde el final en lugar

de desde el principio, existe un método similar llamado `lastIndexOf`:

```
console.log([1, 2, 3, 2, 1].indexOf(2));  
// → 1  
console.log([1, 2, 3, 2, 1].lastIndexOf(2));  
// → 3
```

Tanto `indexOf` como `lastIndexOf` admiten un segundo argumento opcional que indica dónde comenzar la búsqueda.

Otro método fundamental de los arrays es `slice`, que toma índices de inicio y fin y devuelve un array que solo contiene los elementos entre ellos. El índice de inicio es inclusivo, mientras que el índice de fin es exclusivo.

```
console.log([0, 1, 2, 3, 4].slice(2, 4));  
// → [2, 3]  
console.log([0, 1, 2, 3, 4].slice(2));  
// → [2, 3, 4]
```

Cuando no se proporciona el índice de fin, `slice` tomará todos los elementos después del índice de inicio. También puedes omitir el índice de inicio para copiar todo el array.

El método `concat` se puede usar para concatenar arrays y crear un nuevo array, similar a lo que el operador `+` hace para las strings.

El siguiente ejemplo muestra tanto `concat` como `slice` en acción. Toma un array y un índice y devuelve un nuevo array que es una copia del array original con el elemento en el índice dado eliminado:

```
function remove(array, index) {  
    return array.slice(0, index)
```

```
        .concat(array.slice(index + 1));
    }
    console.log(remove(["a", "b", "c", "d", "e"], 2));
    // → ["a", "b", "d", "e"]
```

Si le pasas a `concat` un argumento que no es un array, ese valor se agregará al nuevo array como si fuera un array de un solo elemento.

STRINGS Y SUS PROPIEDADES

Podemos acceder a propiedades como `length` y `toUpperCase` en valores de tipo string. Pero si intentamos añadir una nueva propiedad, esta no se conserva.

```
let kim = "Kim";
kim.age = 88;
console.log(kim.age);
// → undefined
```

Los valores de tipo string, number y Boolean no son objetos, y aunque el lenguaje no se queja si intentas establecer nuevas propiedades en ellos, en realidad no almacena esas propiedades. Como se mencionó anteriormente, dichos valores son inmutables y no pueden ser modificados.

Pero estos tipos tienen propiedades integradas. Cada valor string tiene varios métodos. Algunos muy útiles son `slice` e `indexOf`, que se parecen a los métodos de arrays del mismo nombre:

```
console.log("coconuts".slice(4, 7));
// → nut
console.log("coconut".indexOf("u"));
// → 5
```


Una diferencia es que el `indexOf` de un string puede buscar un string que contenga más de un carácter, mientras que el método correspondiente de arrays busca solo un elemento:

```
console.log("one two three".indexOf("ee"));  
// → 11
```

El método `trim` elimina los espacios en blanco (espacios, saltos de línea, tabulaciones y caracteres similares) del principio y final de una cadena:

```
console.log("  okay \n ".trim());  
// → okay
```

La función `zeroPad` del [capítulo anterior](#) también existe como un método. Se llama `padStart` y recibe la longitud deseada y el carácter de relleno como argumentos:

```
console.log(String(6).padStart(3, "0"));  
// → 006
```

Puedes dividir una cadena en cada ocurrencia de otra cadena con `split` y unirla nuevamente con `join`:

```
let sentence = "Secretarybirds specialize in stomping";  
let words = sentence.split(" ");  
console.log(words);  
// → ["Secretarybirds", "specialize", "in", "stomping"]  
console.log(words.join(". "));  
// → Secretarybirds. specialize. in. stomping
```

Una cadena puede repetirse con el método `repeat`, que crea una nueva cadena que contiene múltiples copias de la cadena original, pegadas juntas:

```
console.log("LA".repeat(3));  
// → LALALA
```

Ya hemos visto la propiedad `length` del tipo `string`. Acceder a los caracteres individuales en una cadena se parece a acceder a los elementos de un array (con una complicación que discutiremos en [Capítulo 5](#)).

```
let string = "abc";  
console.log(string.length);  
// → 3  
console.log(string[1]);  
// → b
```

PARÁMETROS RESTANTES

Puede ser útil para una función aceptar cualquier cantidad de argumento). Por ejemplo, `Math.max` calcula el máximo de *todos* los argumentos que se le pasan. Para escribir una función así, colocas tres puntos antes del último parámetro de la función, de esta manera:

```
function max(...numbers) {  
  let result = -Infinity;  
  for (let number of numbers) {  
    if (number > result) result = number;  
  }  
  return result;  
}  
console.log(max(4, 1, 9, -2));  
// → 9
```

Cuando se llama a una función así, el *parámetro restante* se vincula a un array que contiene todos los argumentos restantes. Si hay otros parámetros antes de él, sus valores no forman parte de ese array.

Cuando, como en `max`, es el único parámetro, contendrá todos los argumentos.

Puedes usar una notación similar de tres puntos para *llamar* a una función con un array de argumentos:

```
let numbers = [5, 1, 7];
console.log(max(...numbers));
// → 7
```

Esto “expande” el array en la llamada de la función, pasando sus elementos como argumentos separados. Es posible incluir un array de esa manera junto con otros argumentos, como en `max(9, ...numbers, 2)`.

La notación de array entre corchetes cuadrados permite al operador de triple punto expandir otro array en el nuevo array:

```
let words = ["never", "fully"];

console.log(["will", ...words, "understand"]);
// → ["will", "never", "fully", "understand"]
```

Esto funciona incluso en objetos con llaves, donde agrega todas las propiedades de otro objeto. Si una propiedad se agrega varias veces, el último valor añadido es el que se conserva:

```
let coordenadas = {x: 10, y: 0};
console.log({...coordenadas, y: 5, z: 1});
// → {x: 10, y: 5, z: 1}
```

EL OBJETO MATH

Como hemos visto, `Math` es una bolsa de funciones de utilidad relacionadas con números, tales como `Math.max` (máximo), `Math.min` (mínimo) y `Math.sqrt` (raíz cuadrada).

El objeto `Math` se utiliza como un contenedor para agrupar un conjunto de funcionalidades relacionadas. Solo hay un objeto `Math` y casi nunca es útil como un valor. Más bien, proporciona un *espacio de nombres* para que todas estas funciones y valores no tengan que ser enlaces globales.

Tener demasiados enlaces globales “contamina” el espacio de nombres. Cuantos más nombres se hayan tomado, más probable es que sobrescribas accidentalmente el valor de algún enlace existente. Por ejemplo, es probable que quieras nombrar algo `max` en uno de tus programas. Dado que la función `max` integrada de JavaScript está protegida de forma segura dentro del objeto `Math`, no tienes que preocuparte por sobrescribirla.

Muchos lenguajes te detendrán, o al menos te advertirán, cuando estés definiendo un enlace con un nombre que ya está tomado. JavaScript hace esto para enlaces que declaraste con `let` o `const`, pero —perversamente— no para enlaces estándar ni para enlaces declarados con `var` o `function`.

Volviendo al objeto `Math`. Si necesitas hacer trigonometría, `Math` puede ayudarte. Contiene `cos` (coseno), `sin` (seno) y `tan` (tangente), así como sus funciones inversas, `acos`, `asin` y `atan`, respectivamente. El número π (pi) —o al menos la aproximación más cercana que cabe en un número de JavaScript— está disponible como `Math.PI`. Existe una antigua tradición de programación que

consiste en escribir los nombres de valores constantes en mayúsculas:

```
function puntoAleatorioEnCirculo(radio) {  
  let ángulo = Math.random() * 2 * Math.PI;  
  return {x: radio * Math.cos(ángulo),  
          y: radio * Math.sin(ángulo)};  
}  
console.log(puntoAleatorioEnCirculo(2));  
// → {x: 0.3667, y: 1.966}
```

Si no estás familiarizado con senos y cosenos, no te preocupes. Los explicaré cuando se utilicen en este libro, en [Capítulo 14](#).

El ejemplo anterior utilizó `Math.random`. Esta es una función que devuelve un nuevo número pseudoaleatorio entre cero (inclusive) y uno (exclusivo) cada vez que la llamas:

```
console.log(Math.random());  
// → 0.36993729369714856  
console.log(Math.random());  
// → 0.727367032552138  
console.log(Math.random());  
// → 0.40180766698904335
```

Aunque las computadoras son máquinas deterministas —siempre reaccionan de la misma manera si se les da la misma entrada— es posible hacer que produzcan números que parezcan aleatorios. Para lograrlo, la máquina mantiene algún valor oculto y, cada vez que solicitas un nuevo número aleatorio, realiza cálculos complicados en este valor oculto para crear un valor nuevo. Almacena un nuevo valor y devuelve algún número derivado de este. De esta manera, puede producir números nuevos y difíciles de predecir que se *aparentan* aleatorios.

Si queremos un número entero aleatorio en lugar de uno fraccionario, podemos usar `Math.floor` (que redondea hacia abajo al número entero más cercano) en el resultado de `Math.random`:

```
console.log(Math.floor(Math.random() * 10));  
// → 2
```

Al multiplicar el número aleatorio por 10, obtenemos un número mayor o igual a 0 y menor que 10. Dado que `Math.floor` redondea hacia abajo, esta expresión producirá, con igual probabilidad, cualquier número del 0 al 9.

También existen las funciones `Math.ceil` (para “techo”, que redondea hacia arriba al número entero más cercano), `Math.round` (al número entero más cercano) y `Math.abs`, que toma el valor absoluto de un número, es decir, niega los valores negativos pero deja los positivos tal como están.

DESESTRUCTURACIÓN

Volviendo por un momento a la función `phi`.

```
function phi(table) {  
  return (table[3] * table[0] - table[2] * table[1]) /  
    Math.sqrt((table[2] + table[3]) *  
      (table[0] + table[1]) *  
      (table[1] + table[3]) *  
      (table[0] + table[2]));  
}
```

Una razón por la que esta función es difícil de leer es que tenemos una asignación apuntando a nuestro array, pero preferiríamos tener asignaciones para los *elementos* del array, es decir, `let n00 =`

`table[0]` y así sucesivamente. Afortunadamente, hay una forma concisa de hacer esto en JavaScript:

```
function phi([n00, n01, n10, n11]) {  
  return (n11 * n00 - n10 * n01) /  
    Math.sqrt((n10 + n11) * (n00 + n01) *  
      (n01 + n11) * (n00 + n10));  
}
```

Esto también funciona para asignaciones creadas con `let`, `var` o `const`. Si sabes que el valor que estás asignando es un array, puedes usar corchetes para “mirar dentro” del valor y asignar sus contenidos.

Un truco similar funciona para objetos, usando llaves en lugar de corchetes:

```
let {name} = {name: "Faraji", age: 23};  
console.log(name);  
// → Faraji
```

Ten en cuenta que si intentas desestructurar `null` o `undefined`, obtendrás un error, igual que si intentarás acceder directamente a una propiedad de esos valores.

ACCESO OPCIONAL A PROPIEDADES

Cuando no estás seguro de si un valor dado produce un objeto pero aún deseas leer una propiedad de él cuando lo hace, puedes usar una variante de la notación de punto: `objeto?.propiedad`.

```
function city(objeto) {  
  return objeto.address?.city;  
}
```

```
console.log(city({address: {city: "Toronto"}}));  
// → Toronto  
console.log(city({name: "Vera"}));  
// → undefined
```

La expresión `a?.b` significa lo mismo que `a.b` cuando `a` no es nulo o indefinido. Cuando lo es, se evalúa como indefinido. Esto puede ser conveniente cuando, como en el ejemplo, no estás seguro de si una propiedad dada existe o cuando una variable podría contener un valor indefinido.

Una notación similar se puede utilizar con el acceso a corchetes cuadrados, e incluso con llamadas de funciones, colocando `?.` delante de los paréntesis o corchetes:

```
console.log("string".notAMethod?.());  
// → undefined  
console.log({}.arrayProp?.[0]);  
// → undefined
```

JSON

Debido a que las propiedades capturan su valor en lugar de contenerlo, los objetos y arrays se almacenan en la memoria de la computadora como secuencias de bits que contienen las *direcciones* —el lugar en la memoria— de sus contenidos. Un array con otro array dentro de él consiste en (al menos) una región de memoria para el array interno y otra para el array externo, que contiene (entre otras cosas) un número que representa la dirección del array interno.

Si deseas guardar datos en un archivo para más tarde o enviarlos a otra computadora a través de la red, debes convertir de alguna manera estas marañas de direcciones de memoria en una descripción

que se pueda almacenar o enviar. Podrías enviar toda la memoria de tu computadora junto con la dirección del valor que te interesa, supongo, pero eso no parece ser el mejor enfoque.

Lo que podemos hacer es *serializar* los datos. Eso significa que se convierten en una descripción plana. Un formato de serialización popular se llama *JSON* (pronunciado “Jason”), que significa JavaScript Object Notation. Se utiliza ampliamente como formato de almacenamiento y comunicación de datos en la Web, incluso en lenguajes que no son JavaScript.

JSON se parece al formato de escritura de arrays y objetos de JavaScript, con algunas restricciones. Todos los nombres de propiedades deben estar rodeados de comillas dobles y solo se permiten expresiones de datos simples—no llamadas a funciones, enlaces, o cualquier cosa que implique cálculos reales. Los comentarios no están permitidos en JSON.

Una entrada de diario podría verse así cuando se representa como datos JSON:

```
{  
  "squirrel": false,  
  "events": ["work", "touched tree", "pizza", "running"]  
}
```

JavaScript nos proporciona las funciones `JSON.stringify` y `JSON.parse` para convertir datos a este formato y desde este formato. La primera toma un valor de JavaScript y devuelve una cadena codificada en JSON. La segunda toma dicha cadena y la convierte en el valor que codifica:

```
let string = JSON.stringify({squirrel: false,
                             events: ["weekend"]});

console.log(string);
// → {"squirrel":false,"events":["weekend"]}
console.log(JSON.parse(string).events);
// → ["weekend"]
```

RESUMEN

Los objetos y arrays proporcionan formas de agrupar varios valores en un único valor. Esto nos permite poner un montón de cosas relacionadas en una bolsa y correr con la bolsa en lugar de envolver nuestros brazos alrededor de cada una de las cosas individuales e intentar sostenerlas por separado.

La mayoría de los valores en JavaScript tienen propiedades, con las excepciones siendo `null` y `undefined`. Las propiedades se acceden usando `valor.prop` o `valor["prop"]`. Los objetos tienden a usar nombres para sus propiedades y almacenan más o menos un conjunto fijo de ellas. Los arrays, por otro lado, suelen contener cantidades variables de valores conceptualmente idénticos y usan números (comenzando desde 0) como los nombres de sus propiedades.

Sí *hay* algunas propiedades nombradas en arrays, como `length` y varios métodos. Los métodos son funciones que viven en propiedades y (usualmente) actúan sobre el valor del cual son una propiedad.

Puedes iterar sobre arrays usando un tipo especial de bucle `for`: `for (let elemento of array)`.

EJERCICIOS

LA SUMA DE UN RANGO

La [introducción](#) de este libro insinuó lo siguiente como una forma agradable de calcular la suma de un rango de números:

```
console.log(sum(range(1, 10)));
```

Escribe una función `range` que tome dos argumentos, `inicio` y `fin`, y devuelva un array que contenga todos los números desde `inicio` hasta `fin`, incluyendo `fin`.

Luego, escribe una función `sum` que tome un array de números y devuelva la suma de estos números. Ejecuta el programa de ejemplo y verifica si realmente devuelve 55.

Como asignación adicional, modifica tu función `range` para que tome un tercer argumento opcional que indique el valor de “paso” utilizado al construir el array. Si no se proporciona un paso, los elementos deberían aumentar en incrementos de uno, correspondiendo al comportamiento anterior. La llamada a la función `range(1, 10, 2)` debería devolver `[1, 3, 5, 7, 9]`. Asegúrate de que esto también funcione con valores de paso negativos, de modo que `range(5, 2, -1)` produzca `[5, 4, 3, 2]`.

REVERSIÓN DE UN ARRAY

Los arrays tienen un método `reverse` que cambia el array invirtiendo el orden en el que aparecen sus elementos. Para este ejercicio, escribe dos funciones, `reverseArray` y

`reverseArrayInPlace`. La primera, `reverseArray`, debería tomar un array como argumento y producir un *nuevo* array que tenga los mismos elementos en orden inverso. La segunda, `reverseArrayInPlace`, debería hacer lo que hace el método `reverse`: *modificar* el array dado como argumento invirtiendo sus elementos. Ninguna de las funciones puede utilizar el método `reverse` estándar.

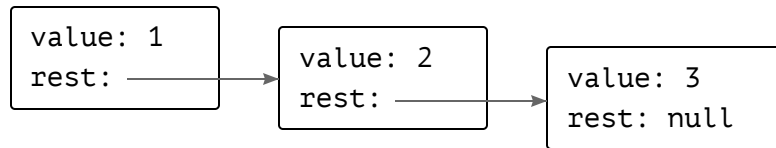
Recordando las notas sobre efectos secundarios y funciones puras en el [capítulo anterior](#), ¿qué variante esperas que sea útil en más situaciones? ¿Cuál se ejecuta más rápido?

LISTA

Como bloques genéricos de valores, los objetos se pueden utilizar para construir todo tipo de estructuras de datos. Una estructura de datos común es la *lista* (no confundir con arrays). Una lista es un conjunto anidado de objetos, donde el primer objeto contiene una referencia al segundo, el segundo al tercero, y así sucesivamente:

```
let list = {  
  value: 1,  
  rest: {  
    value: 2,  
    rest: {  
      value: 3,  
      rest: null  
    }  
  }  
};
```

Los objetos resultantes forman una cadena, como se muestra en el siguiente diagrama:



Una ventaja de las listas es que pueden compartir partes de su estructura. Por ejemplo, si creo dos nuevos valores `{value: 0, rest: list}` y `{value: -1, rest: list}` (siendo `list` la referencia definida anteriormente), son listas independientes, pero comparten la estructura que conforma sus últimos tres elementos. La lista original también sigue siendo válida como una lista de tres elementos.

Escribe una función `arrayToList` que construya una estructura de lista como la mostrada cuando se le da `[1, 2, 3]` como argumento. También escribe una función `listToArray` que produzca un array a partir de una lista. Agrega las funciones auxiliares `prepend`, que toma un elemento y una lista y crea una nueva lista que añade el elemento al principio de la lista de entrada, y `nth`, que toma una lista y un número y devuelve el elemento en la posición dada en la lista (siendo cero el primer elemento) o `undefined` cuando no hay tal elemento.

Si aún no lo has hecho, escribe también una versión recursiva de `nth`.

COMPARACIÓN PROFUNDA

El operador `==` compara objetos por identidad, pero a veces preferirías comparar los valores de sus propiedades reales.

Escribe una función `deepEqual` que tome dos valores y devuelva `true` solo si son el mismo valor o son objetos con las mismas propiedades, donde los valores de las propiedades son iguales cuando se comparan con una llamada recursiva a `deepEqual`.

Para saber si los valores deben compararse directamente (usando el operador `===` para eso) o si sus propiedades deben compararse, puedes usar el operador `typeof`. Si produce `"object"` para ambos valores, deberías hacer una comparación profunda. Pero debes tener en cuenta una excepción tonta: debido a un accidente histórico, `typeof null` también produce `"object"`.

La función `Object.keys` será útil cuando necesites recorrer las propiedades de los objetos para compararlas.

FUNCIONES DE ORDEN SUPERIOR

“Hay dos formas de construir un diseño de software: Una forma es hacerlo tan simple que obviamente no haya deficiencias, y la otra forma es hacerlo tan complicado que no haya deficiencias obvias.”

— C.A.R. Hoare, *Discurso de Recepción del Premio Turing de la ACM de 1980*

Un programa grande es un programa costoso, y no solo por el tiempo que lleva construirlo. El tamaño casi siempre implica complejidad, y la complejidad confunde a los programadores. Los programadores confundidos, a su vez, introducen errores (*bugs*) en los programas. Un programa grande proporciona mucho espacio para que estos errores se escondan, lo que los hace difíciles de encontrar.

Volviendo brevemente a los dos ejemplos finales de programas en la introducción. El primero es autocontenido y tiene seis líneas:

```
let total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
```

El segundo depende de dos funciones externas y tiene una línea:

```
console.log(sum(range(1, 10)));
```

¿Cuál es más probable que contenga un error?

Si contamos el tamaño de las definiciones de `sum` y `range`, el segundo programa también es grande, incluso más que el primero. Pero, aún así, argumentaría que es más probable que sea correcto.

Esto se debe a que la solución se expresa en un vocabulary que corresponde al problema que se está resolviendo. Sumar un rango de números no se trata de bucles y contadores. Se trata de rangos y sumas.

Las definiciones de este vocabulario (las funciones `sum` y `range`) seguirán involucrando bucles, contadores y otros detalles incidentales. Pero debido a que expresan conceptos más simples que el programa en su totalidad, son más fáciles de hacer correctamente.

ABSTRACCIÓN

En el contexto de la programación, este tipo de vocabularios se suelen llamar *abstractions*. Las abstracciones nos brindan la capacidad de hablar sobre problemas a un nivel superior (o más abstracto), sin distraernos con detalles no interesantes.

Como analogía, compara estas dos recetas de sopa de guisantes. La primera es así:

—”Pon 1 taza de guisantes secos por persona en un recipiente. Agrega agua hasta que los guisantes estén bien cubiertos. Deja los guisantes en agua durante al menos 12 horas. Saca los guisantes del agua y ponlos en una olla. Agrega 4 tazas de agua por persona. Cubre la olla y deja que los guisantes hiervan a fuego lento durante dos horas. Toma media cebolla por persona. Córtala en trozos con un cuchillo. Agrégala a los guisantes. Toma un tallo de apio por persona. Córtalo

en trozos con un cuchillo. Agrégalo a los guisantes. Toma una zanahoria por persona. ¡Córtala en trozos! ¡Con un cuchillo! Agrégala a los guisantes. Cocina durante 10 minutos más.”_Cita:

Y esta es la segunda receta:

Por persona: 1 taza de guisantes partidos secos, 4 tazas de agua, media cebolla picada, un tallo de apio y una zanahoria.

Remoja los guisantes durante 12 horas. Cocina a fuego lento durante 2 horas. Pica y agrega las verduras. Cocina durante 10 minutos más.

El segundo es más corto y más fácil de interpretar. Pero necesitas entender algunas palabras más relacionadas con la cocina, como *remoj*ar, *cocinar a fuego lento*, *picar*, y, supongo, *verdura*.

Cuando se programa, no podemos depender de que todas las palabras que necesitamos estén esperándonos en el diccionario. Por lo tanto, podríamos caer en el patrón de la primera receta: trabajar en los pasos precisos que la computadora tiene que realizar, uno por uno, ciegos a los conceptos de más alto nivel que expresan.

Abstraer la repetición

Las funciones simples, como las hemos visto hasta ahora, son una buena manera de construir abstracciones. Pero a veces se quedan cortas.

Es común que un programa haga algo un número determinado de veces. Puedes escribir un `for` para eso, así:

```
for (let i = 0; i < 10; i++) {  
  console.log(i);  
}
```

```
}
```

¿Podemos abstraer “hacer algo N veces” como una función? Bueno, es fácil escribir una función que llame a `console.log` N veces:

```
function repeatLog(n) {  
  for (let i = 0; i < n; i++) {  
    console.log(i);  
  }  
}
```

¿Y si queremos hacer algo que no sea solo registrar los números? Dado que “hacer algo” se puede representar como una función y las funciones son solo valores, podemos pasar nuestra acción como un valor de función:

```
function repeat(n, action) {  
  for (let i = 0; i < n; i++) {  
    action(i);  
  }  
}
```

```
repeat(3, console.log);  
// → 0  
// → 1  
// → 2
```

No tenemos que pasar una función predefinida a `repeat`. A menudo, es más fácil crear un valor de función en el momento:

```
let etiquetas = [];  
repeat(5, i => {  
  etiquetas.push(`Unidad ${i + 1}`);  
});  
console.log(etiquetas);  
// → ["Unidad 1", "Unidad 2", "Unidad 3", "Unidad 4", "Unidad 5"]
```

Esto está estructurado un poco como un `for` loop: primero describe el tipo de loop y luego proporciona un cuerpo. Sin embargo, el cuerpo ahora está escrito como un valor de función, que está envuelto entre los paréntesis de la llamada a `repeat`. Por eso tiene que cerrarse con el corchete de cierre y el paréntesis de cierre. En casos como este ejemplo donde el cuerpo es una sola expresión pequeña, también podrías omitir los corchetes y escribir el bucle en una sola línea.

Funciones de orden superior

Las funciones que operan en otras funciones, ya sea tomándolas como argumentos o devolviéndolas, se llaman *funciones de orden superior*. Dado que ya hemos visto que las funciones son valores regulares, no hay nada particularmente notable sobre el hecho de que existan tales funciones. El término proviene de las matemáticas, donde se toma más en serio la distinción entre funciones y otros valores.

Las funciones de orden superior nos permiten abstraer sobre *acciones*, no solo sobre valores. Vienen en varias formas. Por ejemplo, podemos tener funciones que crean nuevas funciones:

```
function mayorQue(n) {  
  return m => m > n;  
}  
let mayorQue10 = mayorQue(10);  
console.log(mayorQue10(11));  
// → true
```

También podemos tener funciones que modifican otras funciones:

```
function ruidosa(f) {
  return (...args) => {
    console.log("llamando con", args);
    let resultado = f(...args);
    console.log("llamado con", args, ", devolvió", resultado);
    return resultado;
  };
}
ruidosa(Math.min)(3, 2, 1);
// → llamando con [3, 2, 1]
// → llamado con [3, 2, 1] , devolvió 1
```

Incluso podemos escribir funciones que proveen nuevos tipos de flujo de control:

```
function aMenosQue(prueba, entonces) {
  if (!prueba) entonces();
}

repetir(3, n => {
  aMenosQue(n % 2 == 1, () => {
    console.log(n, "es par");
  });
});
// → 0 es par
// → 2 es par
```

Existe un método incorporado de arrays, `forEach`, que proporciona algo similar a un bucle `for/of` como una función de orden superior:

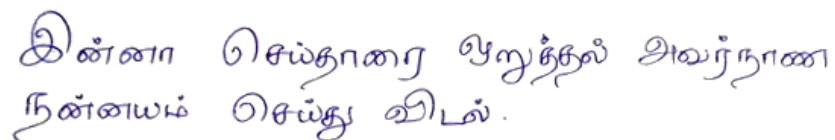
```
["A", "B"].forEach(l => console.log(l));
// → A
// → B
```

CONJUNTO DE DATOS DE SCRIPT

Un área donde las funciones de orden superior destacan es en el procesamiento de datos. Para procesar datos, necesitaremos algunos ejemplos de datos reales. Este capítulo utilizará un conjunto de datos sobre scripts—sistemas de escritura tales como el latín, cirílico o árabe.

¿Recuerdas Unicode del [Capítulo ?](#), el sistema que asigna un número a cada carácter en lenguaje escrito? La mayoría de estos caracteres están asociados con un script específico. El estándar contiene 140 scripts diferentes, de los cuales 81 aún se utilizan hoy en día y 59 son históricos.

Aunque solo puedo leer con fluidez caracteres latinos, aprecio el hecho de que las personas estén escribiendo textos en al menos otros 80 sistemas de escritura, muchos de los cuales ni siquiera reconocería. Por ejemplo, aquí tienes una muestra de escritura Tamil:



இணை செத்தாகர முருத்தல் சிவநாண
நுனாயம் செய்து விடல்.

El ejemplo del conjunto de datos contiene algunas piezas de información sobre los 140 scripts definidos en Unicode. Está disponible en el [sandbox de código](https://eloquentjavascript.net/code#5) para este capítulo (<https://eloquentjavascript.net/code#5>) como el enlace SCRIPTS. El enlace contiene un array de objetos, cada uno describe un script:

```
{  
  name: "Copto",  
  rangos: [[994, 1008], [11392, 11508], [11513, 11520]],  
  dirección: "ltr",
```

```
año: -200,  
vivo: false,  
enlace: "https://es.wikipedia.org/wiki/Alfabeto_copto"  
}
```

Tal objeto nos informa sobre el nombre del script, los rangos Unicode asignados a él, la dirección en la que se escribe, el tiempo de origen (aproximado), si todavía se utiliza, y un enlace a más información. La dirección puede ser "ltr" para izquierda a derecha, "rtl" para derecha a izquierda (como se escribe el texto en árabe y hebreo) o "ttb" para arriba hacia abajo (como en la escritura mongola).

La propiedad `ranges` contiene una matriz de rangos de caracteres Unicode, cada uno de los cuales es una matriz de dos elementos que contiene un límite inferior y un límite superior. Todos los códigos de caracteres dentro de estos rangos se asignan al guion. El límite inferior es inclusivo (el código 994 es un carácter copto) y el límite superior no es inclusivo (el código 1008 no lo es).

FILTRADO DE ARRAYS

Si queremos encontrar los guiones en el conjunto de datos que todavía se utilizan, la siguiente función puede ser útil. Filtra los elementos de una matriz que no pasan una prueba.

```
function filter(array, test) {  
  let passed = [];  
  for (let element of array) {  
    if (test(element)) {  
      passed.push(element);  
    }  
  }  
  return passed;  
}
```

```
}
```

```
console.log(filter(SCRIPTS, script => script.living));  
// → [{name: "Adlam", ...}, ...]
```

La función utiliza el argumento llamado `test`, un valor de función, para llenar un “vacío” en la computación, el proceso de decidir qué elementos recopilar.

Observa cómo la función `filter`, en lugar de eliminar elementos de la matriz existente, construye una nueva matriz con solo los elementos que pasan la prueba. Esta función es *pura*. No modifica la matriz que se le pasa.

Al igual que `forEach`, `filter` es un método de matriz estándar. El ejemplo definió la función solo para mostrar qué hace internamente. De ahora en adelante, lo usaremos de esta manera en su lugar:

```
console.log(SCRIPTS.filter(s => s.direction == "ttb"));  
// → [{name: "Mongolian", ...}, ...]
```

TRANSFORMACIÓN CON MAP

Digamos que tenemos una matriz de objetos que representan guiones, producida al filtrar la matriz `SCRIPTS` de alguna manera. Queremos una matriz de nombres en su lugar, que es más fácil de inspeccionar.

El método `map` transforma una matriz aplicando una función a todos sus elementos y construyendo una nueva matriz a partir de los valores devueltos. La nueva matriz tendrá la misma longitud que la matriz de entrada, pero su contenido habrá sido *mapeado* a una nueva forma por la función:

```
function map(array, transform) {
  let mapped = [];
  for (let element of array) {
    mapped.push(transform(element));
  }
  return mapped;
}

let rtlScripts = SCRIPTS.filter(s => s.direction == "rtl");
console.log(map(rtlScripts, s => s.name));
// → ["Adlam", "Arabic", "Imperial Aramaic", ...]
```

Al igual que `forEach` y `filter`, `map` es un método de matriz estándar.

RESUMEN CON REDUCE

Otra cosa común que hacer con matrices es calcular un único valor a partir de ellas. Nuestro ejemplo recurrente, sumar una colección de números, es una instancia de esto. Otro ejemplo es encontrar el guion con más caracteres.

La operación de orden superior que representa este patrón se llama *reduce* (a veces también llamada *fold*). Construye un valor tomando repetidamente un único elemento del array y combinándolo con el valor actual. Al sumar números, comenzarías con el número cero y, para cada elemento, lo sumarías al total.

Los parámetros de `reduce` son, además del array, una función de combinación y un valor inicial. Esta función es un poco menos directa que `filter` y `map`, así que obsérvala detenidamente:

```
function reduce(array, combine, start) {
  let current = start;
  for (let element of array) {
```



```

    current = combine(current, element);
  }
  return current;
}

console.log(reduce([1, 2, 3, 4], (a, b) => a + b, 0));
// → 10

```

El método estándar de arrays `reduce`, que por supuesto corresponde a esta función, tiene una conveniencia adicional. Si tu array contiene al menos un elemento, puedes omitir el argumento `start`. El método tomará el primer elemento del array como su valor inicial y comenzará a reducir en el segundo elemento.

```

console.log([1, 2, 3, 4].reduce((a, b) => a + b));
// → 10

```

Para usar `reduce` (dos veces) y encontrar el script con más caracteres, podemos escribir algo así:

```

function characterCount(script) {
  return script.ranges.reduce((count, [from, to]) => {
    return count + (to - from);
  }, 0);
}

console.log(SCRIPTS.reduce((a, b) => {
  return characterCount(a) < characterCount(b) ? b : a;
}));
// → {name: "Han", ...}

```

La función `characterCount` reduce los rangos asignados a un script sumando sus tamaños. Observa el uso de la desestructuración en la lista de parámetros de la función reductora. La segunda llamada a `reduce` luego utiliza esto para encontrar el script más

grande comparando repetidamente dos scripts y devolviendo el más grande.

El script Han tiene más de 89,000 caracteres asignados en el estándar Unicode, convirtiéndolo en el sistema de escritura más grande en el conjunto de datos. Han es un script a veces utilizado para texto en chino, japonés y coreano. Esos idiomas comparten muchos caracteres, aunque tienden a escribirlos de manera diferente. El Consorcio Unicode (con sede en EE. UU.) decidió tratarlos como un único sistema de escritura para ahorrar códigos de caracteres. Esto se llama *unificación Han* y todavía molesta a algunas personas.

COMPOSABILIDAD

Considera cómo hubiéramos escrito el ejemplo anterior (encontrando el script más grande) sin funciones de orden superior. El código no es mucho peor:

```
let biggest = null;
for (let script of SCRIPTS) {
  if (biggest == null ||
      characterCount(biggest) < characterCount(script)) {
    biggest = script;
  }
}
console.log(biggest);
// → {name: "Han", ...}
```

Hay algunas variables adicionales y el programa tiene cuatro líneas más, pero sigue siendo muy legible.

Las abstracciones proporcionadas por estas funciones brillan realmente cuando necesitas *componer* operaciones. Como ejemplo, escribamos un código que encuentre el año promedio de origen para scripts vivos y muertos en el conjunto de datos:

```
function average(array) {
  return array.reduce((a, b) => a + b) / array.length;
}

console.log(Math.round(average(
  SCRIPTS.filter(s => s.living).map(s => s.year))));
// → 1165
console.log(Math.round(average(
  SCRIPTS.filter(s => !s.living).map(s => s.year))));
// → 204
```

Como puedes ver, los scripts muertos en Unicode son, en promedio, más antiguos que los vivos. Esta no es una estadística muy significativa o sorprendente. Pero espero que estés de acuerdo en que el código utilizado para calcularlo no es difícil de leer. Puedes verlo como un pipeline: empezamos con todos los scripts, filtramos los vivos (o muertos), tomamos los años de esos scripts, calculamos el promedio y redondeamos el resultado.

Definitivamente también podrías escribir este cálculo como un único loop grande:

```
let total = 0, count = 0;
for (let script of SCRIPTS) {
  if (script.living) {
    total += script.year;
    count += 1;
  }
}
console.log(Math.round(total / count));
// → 1165
```

Sin embargo, es más difícil ver qué se estaba calculando y cómo. Y debido a que los resultados intermedios no se representan como valores coherentes, sería mucho más trabajo extraer algo como `average` en una función separada.

En términos de lo que realmente está haciendo la computadora, estos dos enfoques también son bastante diferentes. El primero construirá nuevos arrays al ejecutar `filter` y `map`, mientras que el segundo calcula solo algunos números, haciendo menos trabajo. Por lo general, puedes permitirte el enfoque legible, pero si estás procesando matrices enormes y haciéndolo muchas veces, el estilo menos abstracto podría valer la pena por la velocidad adicional.

CADENAS Y CÓDIGOS DE CARACTERES

Un uso interesante de este conjunto de datos sería averiguar qué script está utilizando un fragmento de texto. Vamos a través de un programa que hace esto.

Recuerda que cada script tiene asociado un array de intervalos de códigos de caracteres. Dado un código de carácter, podríamos usar una función como esta para encontrar el script correspondiente (si lo hay):

```
function characterScript(code) {
  for (let script of SCRIPTS) {
    if (script.ranges.some(([from, to]) => {
      return code >= from && code < to;
    }))) {
      return script;
    }
  }
  return null;
}
```

```
console.log(characterScript(121));  
// → {name: "Latin", ...}
```

El método `some` es otra función de orden superior. Toma una función de prueba y te dice si esa función devuelve `true` para alguno de los elementos en el array.

Pero, ¿cómo obtenemos los códigos de caracteres en una cadena?

En [Chapter 1](#) mencioné que las cadenas de JavaScript están codificadas como una secuencia de números de 16 bits. Estos se llaman *unidades de código*. Un código de carácter Unicode inicialmente se suponía que cabía dentro de tal unidad (lo que te da un poco más de 65,000 caracteres). Cuando quedó claro que eso no iba a ser suficiente, muchas personas se mostraron reacias a la necesidad de usar más memoria por carácter. Para abordar estas preocupaciones, se inventó UTF-16, el formato también utilizado por las cadenas de JavaScript. Describe la mayoría de los caracteres comunes usando una única unidad de código de 16 bits, pero usa un par de dos unidades de dicho tipo para otros.

UTF-16 generalmente se considera una mala idea hoy en día. Parece casi diseñado intencionalmente para invitar a errores. Es fácil escribir programas que pretendan que las unidades de código y los caracteres son lo mismo. Y si tu lenguaje no utiliza caracteres de dos unidades, eso parecerá funcionar perfectamente. Pero tan pronto como alguien intente usar dicho programa con algunos caracteres chinos menos comunes, fallará. Afortunadamente, con la llegada de los emoji, todo el mundo ha comenzado a usar caracteres de dos

unidades, y la carga de tratar con tales problemas está más equitativamente distribuida.

Lamentablemente, las operaciones obvias en las cadenas de JavaScript, como obtener su longitud a través de la propiedad `length` y acceder a su contenido usando corchetes cuadrados, tratan solo con unidades de código.

```
// Dos caracteres emoji, caballo y zapato
let horseShoe = "🐎👟";
console.log(horseShoe.length);
// → 4
console.log(horseShoe[0]);
// → (Mitad de carácter inválida)
console.log(horseShoe.charCodeAt(0));
// → 55357 (Código de la mitad de carácter)
console.log(horseShoe.codePointAt(0));
// → 128052 (Código real para el emoji de caballo)
```

El método `charCodeAt` de JavaScript te da una unidad de código, no un código de carácter completo. El método `codePointAt`, añadido más tarde, sí da un carácter Unicode completo, por lo que podríamos usarlo para obtener caracteres de una cadena. Pero el argumento pasado a `codePointAt` sigue siendo un índice en la secuencia de unidades de código. Para recorrer todos los caracteres en una cadena, aún necesitaríamos abordar la cuestión de si un carácter ocupa una o dos unidades de código.

En el [capítulo anterior](#), mencioné que un bucle `for/of` también se puede usar en cadenas. Al igual que `codePointAt`, este tipo de bucle se introdujo en un momento en que la gente era muy consciente de los problemas con UTF-16. Cuando lo usas para

recorrer una cadena, te proporciona caracteres reales, no unidades de código:

```
let roseDragon = "🌹🐉";
for (let char of roseDragon) {
  console.log(char);
}
// → 🌹
// → 🐉
```

Si tienes un carácter (que será una cadena de una o dos unidades de código), puedes usar `codePointAt(0)` para obtener su código.

RECONOCIENDO TEXTO

Tenemos una función `characterScript` y una forma de recorrer correctamente los caracteres. El próximo paso es contar los caracteres que pertenecen a cada script. La siguiente abstracción de conteo será útil para eso:

```
function countBy(items, groupName) {
  let counts = [];
  for (let item of items) {
    let name = groupName(item);
    let known = counts.find(c => c.name == name);
    if (!known) {
      counts.push({name, count: 1});
    } else {
      known.count++;
    }
  }
  return counts;
}

console.log(countBy([1, 2, 3, 4, 5], n => n > 2));
// → [{name: false, count: 2}, {name: true, count: 3}]
```

La función `countBy` espera una colección (cualquier cosa por la que podamos iterar con `for/of`) y una función que calcule un nombre de grupo para un elemento dado. Devuelve una matriz de objetos, cada uno de los cuales nombra un grupo y te dice el número de elementos que se encontraron en ese grupo.

Utiliza otro método de array, `find`, que recorre los elementos en el array y devuelve el primero para el cual una función devuelve `true`. Devuelve `undefined` cuando no se encuentra dicho elemento.

Usando `countBy`, podemos escribir la función que nos dice qué scripts se utilizan en un fragmento de texto:

```
function textScripts(text) {
  let scripts = countBy(text, char => {
    let script = characterScript(char.codePointAt(0));
    return script ? script.name : "ninguno";
  }).filter(({name}) => name !== "ninguno");

  let total = scripts.reduce((n, {count}) => n + count, 0);
  if (total === 0) return "No se encontraron scripts";

  return scripts.map(({name, count}) => {
    return `${Math.round(count * 100 / total)}% ${name}`;
  }).join(", ");
}

console.log(textScripts('英国的狗说"woof", 俄罗斯的狗说"ТЯБ"'));
// → 61% Han, 22% Latin, 17% Cyrillic
```

La función primero cuenta los caracteres por nombre, usando `characterScript` para asignarles un nombre y retrocediendo a la cadena "ninguno" para los caracteres que no forman parte de ningún script. La llamada a `filter` elimina la entrada de

"ninguno" del array resultante, ya que no nos interesan esos caracteres.

Para poder calcular porcentajes, primero necesitamos el número total de caracteres que pertenecen a un script, lo cual podemos calcular con `reduce`. Si no se encuentran dichos caracteres, la función devuelve una cadena específica. De lo contrario, transforma las entradas de conteo en cadenas legibles con `map` y luego las combina con `join`.

RESUMEN

Poder pasar valores de funciones a otras funciones es un aspecto muy útil de JavaScript. Nos permite escribir funciones que modelan cálculos con “vacíos”. El código que llama a estas funciones puede llenar los vacíos proporcionando valores de funciones.

Los arrays proporcionan diversos métodos de orden superior útiles. Puedes usar `forEach` para recorrer los elementos de un array. El método `filter` devuelve un nuevo array que contiene solo los elementos que pasan la función de predicado. Transformar un array poniendo cada elemento en una función se hace con `map`. Puedes usar `reduce` para combinar todos los elementos de un array en un único valor. El método `some` comprueba si algún elemento coincide con una función de predicado dada, mientras que `find` encuentra el primer elemento que coincide con un predicado.

EJERCICIOS

APLANAMIENTO

Utiliza el método `reduce` en combinación con el método `concat` para “aplanar” un array de arrays en un único array que contenga todos los elementos de los arrays originales.

TU PROPIO BUCLE

Escribe una función de orden superior `loop` que proporcione algo similar a una declaración `for loop`. Debería recibir un valor, una función de prueba, una función de actualización y una función de cuerpo. En cada iteración, primero debe ejecutar la función de prueba en el valor actual del bucle y detenerse si devuelve falso. Luego debe llamar a la función de cuerpo, dándole el valor actual, y finalmente llamar a la función de actualización para crear un nuevo valor y empezar de nuevo desde el principio.

Al definir la función, puedes usar un bucle regular para hacer el bucle real.

EVERYTHING

Los arrays también tienen un método `every` análogo al método `some`. Este método devuelve `true` cuando la función dada devuelve `true` para *cada* elemento en el array. En cierto modo, `some` es una versión del operador `||` que actúa en arrays, y `every` es como el operador `&&`.

Implementa `every` como una función que recibe un array y una función de predicado como parámetros. Escribe dos versiones, una usando un bucle y otra usando el método `some`.

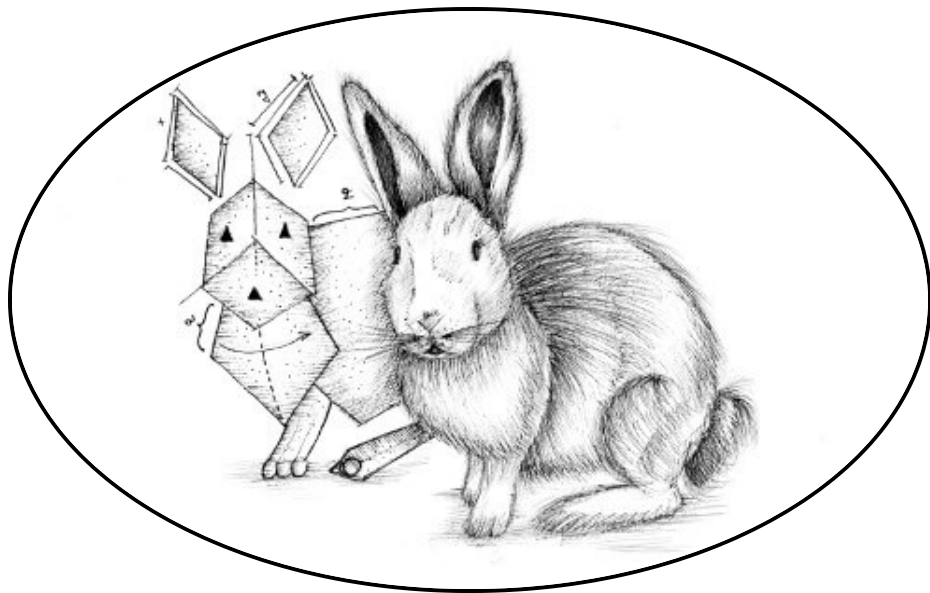
DIRECCIÓN DE ESCRITURA DOMINANTE

Escribe una función que calcule la dirección de escritura dominante en una cadena de texto. Recuerda que cada objeto `script` tiene una propiedad `direction` que puede ser `"ltr"` (de izquierda a derecha), `"rtl"` (de derecha a izquierda) o `"ttb"` (de arriba a abajo).

LA VIDA SECRETA DE LOS OBJETOS

“Un tipo de dato abstracto se realiza escribiendo un tipo especial de programa [...] que define el tipo en términos de las operaciones que se pueden realizar en él.”

—Barbara Liskov, *Programando con Tipos de Datos Abstractos*



[El Capítulo 4](#) introdujo los objetos de JavaScript, como contenedores que almacenan otros datos.

En la cultura de la programación, tenemos algo llamado *programación orientada a objetos*, un conjunto de técnicas que utilizan objetos como principio central de la organización de programas. Aunque nadie realmente se pone de acuerdo en su definición precisa, la programación orientada a objetos ha dado forma al diseño de muchos lenguajes de programación, incluido

JavaScript. Este capítulo describe la forma en que estas ideas se pueden aplicar en JavaScript.

TIPOS DE DATOS ABSTRACTOS

La idea principal en la programación orientada a objetos es utilizar objetos, o más bien *tipos* de objetos, como la unidad de organización del programa. Configurar un programa como una serie de tipos de objetos estrictamente separados proporciona una forma de pensar en su estructura y, por lo tanto, de imponer algún tipo de disciplina para evitar que todo se entrelace.

La forma de hacer esto es pensar en objetos de alguna manera similar a como pensarías en una batidora eléctrica u otro electrodoméstico para el consumidor. Hay personas que diseñaron y ensamblaron una batidora, y tienen que realizar un trabajo especializado que requiere ciencia de materiales y comprensión de la electricidad. Cubren todo eso con una carcasa de plástico suave, de modo que las personas que solo quieren mezclar masa para panqueques no tengan que preocuparse por todo eso, solo tienen que entender los pocos botones con los que se puede operar la batidora.

De manera similar, un tipo de dato abstracto, o clase de objeto, es un subprograma que puede contener un código arbitrariamente complicado, pero expone un conjunto limitado de métodos y propiedades que se supone que las personas que trabajan con él deben usar. Esto permite construir programas grandes a partir de varios tipos de electrodomésticos, limitando el grado en que estas diferentes partes están entrelazadas al requerir que solo interactúen entre sí de formas específicas.

Si se encuentra un problema en una clase de objeto como esta, a menudo se puede reparar, o incluso reescribir completamente, sin afectar el resto del programa.

Incluso mejor, puede ser posible utilizar clases de objetos en varios programas diferentes, evitando la necesidad de recrear su funcionalidad desde cero. Puedes pensar en las estructuras de datos integradas de JavaScript, como arrays y strings, como tipos de datos abstractos reutilizables de este tipo.

Cada tipo de dato abstracto tiene una *interfaz*, que es la colección de operaciones que el código externo puede realizar en él. Incluso cosas básicas como los números pueden considerarse un tipo de dato abstracto cuya interfaz nos permite sumarlos, multiplicarlos, compararlos, y así sucesivamente. De hecho, la fijación en objetos *individuales* como la unidad principal de organización en la programación orientada a objetos clásica es un tanto desafortunada, ya que a menudo las piezas de funcionalidad útiles involucran un grupo de diferentes clases de objetos que trabajan estrechamente juntos.

MÉTODOS

En JavaScript, los métodos no son más que propiedades que contienen valores de función. Este es un método simple:

```
function speak(line) {  
  console.log(`El conejo ${this.type} dice '${line}'`);  
}  
let conejoBlanco = {type: "blanco", speak};  
let conejoHambriento = {type: "hambriento", speak};  
  
conejoBlanco.speak("Oh, mi pelaje y mis bigotes");
```

```
// → El conejo blanco dice 'Oh, mi pelaje y mis bigotes'  
conejoHambriento.speak("¿Tienes zanahorias?");  
// → El conejo hambriento dice '¿Tienes zanahorias?'
```

Típicamente, un método necesita hacer algo con el objeto en el que fue invocado. Cuando una función es llamada como método—buscada como propiedad y llamada inmediatamente, como en `objeto.método()`—la vinculación llamada `this` en su cuerpo apunta automáticamente al objeto en el que fue llamada.

Puedes pensar en `this` como un parámetro extra que se pasa a la función de una manera diferente a los parámetros regulares. Si deseas proveerlo explícitamente, puedes usar el método `call` de una función, el cual toma el valor de `this` como su primer argumento y trata los siguientes argumentos como parámetros normales.

```
speak.call(conejoBlanco, "Rápido");  
// → El conejo blanco dice 'Rápido'
```

Dado que cada función tiene su propia vinculación `this`, cuyo valor depende de la forma en que es llamada, no puedes hacer referencia al `this` del ámbito envolvente en una función regular definida con la palabra clave `function`.

Las funciones flecha son diferentes—no vinculan su propio `this` pero pueden ver la vinculación `this` del ámbito que las rodea. Por lo tanto, puedes hacer algo como el siguiente código, el cual hace referencia a `this` desde dentro de una función local:

```
let buscador = {  
  find(array) {  
    return array.some(v => v == this.value);  
  }  
};
```

```
    },  
    value: 5  
  };  
  console.log(buscador.find([4, 5]));  
  // → true
```

Una propiedad como `find(array)` en una expresión de objeto es una forma abreviada de definir un método. Crea una propiedad llamada `find` y le asigna una función como su valor.

Si hubiera escrito el argumento de `some` usando la palabra clave `function`, este código no funcionaría.

PROTOTIPOS

Entonces, una forma de crear un tipo de conejo abstracto con un método `speak` sería crear una función de ayuda que tenga un tipo de conejo como parámetro, y devuelva un objeto que contenga eso como su propiedad `type` y nuestra función `speak` en su propiedad `speak`.

Todos los conejos comparten ese mismo método. Especialmente para tipos con muchos métodos, sería conveniente tener una forma de mantener los métodos de un tipo en un solo lugar, en lugar de añadirlos a cada objeto individualmente.

En JavaScript, los *prototipos* son la forma de lograr eso. Los objetos pueden estar enlazados a otros objetos, para obtener mágicamente todas las propiedades que ese otro objeto tiene. Los simples objetos creados con la notación `{}` están enlazados a un objeto llamado `Object.prototype`.


```
let empty = {};  
console.log(empty.toString);  
// → function toString(){...}  
console.log(empty.toString());  
// → [object Object]
```

Parece que acabamos de extraer una propiedad de un objeto vacío. Pero de hecho, `toString` es un método almacenado en `Object.prototype`, lo que significa que está disponible en la mayoría de los objetos.

Cuando a un objeto se le solicita una propiedad que no tiene, se buscará en su prototipo la propiedad. Si éste no la tiene, se buscará en su prototipo, y así sucesivamente hasta llegar a un objeto que no tiene prototipo (`Object.prototype` es un objeto de este tipo).

```
console.log(Object.getPrototypeOf({}) == Object.prototype);  
// → true  
console.log(Object.getPrototypeOf(Object.prototype));  
// → null
```

Como podrás imaginar, `Object.getPrototypeOf` devuelve el prototipo de un objeto.

Muchos objetos no tienen directamente `Object.prototype` como su prototipo, sino que tienen otro objeto que proporciona un conjunto diferente de propiedades predeterminadas. Las funciones se derivan de `Function.prototype`, y los arreglos se derivan de `Array.prototype`.

```
console.log(Object.getPrototypeOf(Math.max) ==  
              Function.prototype);  
// → true  
console.log(Object.getPrototypeOf([]) == Array.prototype);  
// → true
```

Un objeto prototipo de este tipo tendrá a su vez un prototipo, a menudo `Object.prototype`, de modo que aún proporciona de forma indirecta métodos como `toString`.

Puedes utilizar `Object.create` para crear un objeto con un prototipo específico.

```
let protoRabbit = {  
  speak(line) {  
    console.log(`El conejo ${this.type} dice '${line}'`);  
  }  
};  
let blackRabbit = Object.create(protoRabbit);  
blackRabbit.type = "negro";  
blackRabbit.speak("Soy el miedo y la oscuridad");  
// → El conejo negro dice 'Soy el miedo y la oscuridad'
```

El conejo “proto” actúa como un contenedor para las propiedades que son compartidas por todos los conejos. Un objeto de conejo individual, como el conejo negro, contiene propiedades que se aplican solo a él mismo, en este caso su tipo, y deriva propiedades compartidas de su prototipo.

CLASES

El sistema de prototipos de JavaScript puede interpretarse como una versión algo libre de los tipos de datos abstractos o clases. Una clase define la forma de un tipo de objeto, los métodos y propiedades que tiene. A dicho objeto se le llama una *instancia* de la clase.

Los prototipos son útiles para definir propiedades cuyo valor es compartido por todas las instancias de una clase. Las propiedades

que difieren por instancia, como la propiedad `type` de nuestros conejos, deben ser almacenadas directamente en los objetos mismos.

Así que para crear una instancia de una clase, debes hacer un objeto que se derive del prototipo adecuado, pero *también* debes asegurarte de que él mismo tenga las propiedades que se supone que deben tener las instancias de esta clase. Esto es lo que hace una función *constructor*.

```
function makeRabbit(type) {  
  let rabbit = Object.create(protoRabbit);  
  rabbit.type = type;  
  return rabbit;  
}
```

La notación de `class` de JavaScript facilita la definición de este tipo de función, junto con un objeto `prototype`.

```
class Rabbit {  
  constructor(type) {  
    this.type = type;  
  }  
  speak(line) {  
    console.log(`El conejo ${this.type} dice '${line}'`);  
  }  
}
```

La palabra clave `class` inicia una declaración de clase, que nos permite definir un constructor y un conjunto de métodos juntos. Se pueden escribir cualquier cantidad de métodos dentro de las llaves de la declaración. Este código tiene el efecto de definir un enlace llamado `Rabbit`, que contiene una función que ejecuta el código en constructor, y tiene una propiedad `prototype` que contiene el método `speak`.

Esta función no puede ser llamada normalmente. Los constructores, en JavaScript, se llaman colocando la palabra clave `new` delante de ellos. Al hacerlo, se crea un objeto nuevo con el objeto contenido en la propiedad `prototype` de la función como prototipo, luego se ejecuta la función con `this` vinculado al nuevo objeto, y finalmente se devuelve el objeto.

```
let killerRabbit = new Rabbit("asesino");
```

De hecho, la palabra clave `class` se introdujo solo en la edición de JavaScript de 2015. Cualquier función puede ser utilizada como constructor, y antes de 2015 la forma de definir una clase era escribir una función regular y luego manipular su propiedad `prototype`.

```
function ConejoArcaico(type) {
  this.type = type;
}

ConejoArcaico.prototype.speak = function(line) {
  console.log(`El conejo ${this.type} dice '${line}'`);
};

let conejoEstiloAntiguo = new ConejoArcaico("estilo antiguo");
```

Por esta razón, todas las funciones que no sean de flecha comienzan con una propiedad `prototype` que contiene un objeto vacío.

Por convención, los nombres de constructores se escriben con mayúscula inicial para que puedan distinguirse fácilmente de otras funciones.

Es importante entender la distinción entre la forma en que un prototipo está asociado con un constructor (a través de su *propiedad*

prototype) y la forma en que los objetos *tienen* un prototipo (que se puede encontrar con `Object.getPrototypeOf()`). El prototipo real de un constructor es `Function.prototype` ya que los constructores son funciones. Su *propiedad* `prototype` contiene el prototipo utilizado para las instancias creadas a través de él.

```
console.log(Object.getPrototypeOf(Rabbit) ==
              Function.prototype);
// → true
console.log(Object.getPrototypeOf(killerRabbit) ==
              Rabbit.prototype);
// → true
```

Por lo general, los constructores agregarán algunas propiedades específicas de instancia a `this`. También es posible declarar propiedades directamente en la declaración de clase. A diferencia de los métodos, dichas propiedades se agregan a los objetos instancia, no al prototipo.

```
class Particle {
  speed = 0;
  constructor(position) {
    this.position = position;
  }
}
```

Al igual que `function`, `class` se puede utilizar tanto en declaraciones como en expresiones. Cuando se usa como una expresión, no define un enlace sino que simplemente produce el constructor como un valor. Se te permite omitir el nombre de la clase en una expresión de clase.

```
let object = new class { getWord() { return "hello"; } };
console.log(object.getWord());
// → hello
```

PROPIEDADES PRIVADAS

Es común que las clases definan algunas propiedades y métodos para uso interno, que no forman parte de su interfaz. Estas se llaman propiedades *privadas*, en contraposición a las públicas, que son parte de la interfaz externa del objeto.

Para declarar un método privado, coloca un signo `#` delante de su nombre. Estos métodos solo pueden ser llamados desde dentro de la declaración de la `class` que los define.

```
class SecretiveObject {
  #getSecret() {
    return "Me comí todas las ciruelas";
  }
  interrogate() {
    let deboDecirlo = this.#getSecret();
    return "nunca";
  }
}
```

Si intentas llamar a `#getSecret` desde fuera de la clase, obtendrás un error. Su existencia está completamente oculta dentro de la declaración de la clase.

Para usar propiedades de instancia privadas, debes declararlas. Las propiedades regulares se pueden crear simplemente asignándoles un valor, pero las propiedades privadas *deben* declararse en la declaración de la clase para estar disponibles en absoluto.

Esta clase implementa un dispositivo para obtener un número entero aleatorio por debajo de un número máximo dado. Solo tiene una propiedad pública: `getNumber`.

```

class RandomSource {
  #max;
  constructor(max) {
    this.#max = max;
  }
  getNumber() {
    return Math.floor(Math.random() * this.#max);
  }
}

```

SOBRESCRIBIENDO PROPIEDADES DERIVADAS

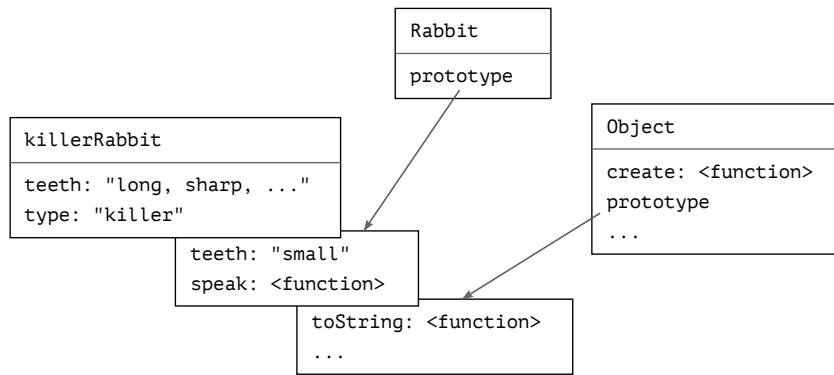
Cuando agregas una propiedad a un objeto, ya sea que esté presente en el prototipo o no, la propiedad se agrega al objeto *mismo*. Si ya existía una propiedad con el mismo nombre en el prototipo, esta propiedad ya no afectará al objeto, ya que ahora está oculta detrás de la propiedad propia del objeto.

```

Rabbit.prototype.teeth = "pequeñas";
console.log(killerRabbit.teeth);
// → pequeñas
killerRabbit.teeth = "largos, afilados y sangrientos";
console.log(killerRabbit.teeth);
// → largos, afilados y sangrientos
console.log((new Rabbit("básico")).teeth);
// → pequeñas
console.log(Rabbit.prototype.teeth);
// → pequeñas

```

El siguiente diagrama esquematiza la situación después de que se ha ejecutado este código. Los prototipos `Rabbit` y `Object` están detrás de `killerRabbit` como un telón de fondo, donde se pueden buscar propiedades que no se encuentran en el objeto mismo.



Sobrescribir propiedades que existen en un prototipo puede ser algo útil de hacer. Como muestra el ejemplo de los dientes del conejo, sobrescribir se puede utilizar para expresar propiedades excepcionales en instancias de una clase más genérica de objetos, mientras se permite que los objetos no excepcionales tomen un valor estándar de su prototipo.

También se utiliza la sobrescritura para dar a los prototipos estándar de funciones y arrays un método `toString` diferente al del prototipo básico de objeto.

```
console.log(Array.prototype.toString ==
              Object.prototype.toString);
// → false
console.log([1, 2].toString());
// → 1,2
```

Llamar a `toString` en un array produce un resultado similar a llamar a `.join(", ")` en él—coloca comas entre los valores en el array. Llamar directamente a `Object.prototype.toString` con un array produce una cadena diferente. Esa función no conoce acerca de los arrays, por lo que simplemente coloca la palabra *object* y el nombre del tipo entre corchetes.


```
console.log(Object.prototype.toString.call([1, 2]));  
// → [object Array]
```

MAPAS

Vimos la palabra *map* utilizada en el [capítulo anterior](#) para una operación que transforma una estructura de datos aplicando una función a sus elementos. Por confuso que sea, en programación la misma palabra también se utiliza para una cosa relacionada pero bastante diferente.

Un *mapa* (sustantivo) es una estructura de datos que asocia valores (las claves) con otros valores. Por ejemplo, podrías querer mapear nombres a edades. Es posible usar objetos para esto.

```
let edades = {  
  Boris: 39,  
  Liang: 22,  
  Júlia: 62  
};  
  
console.log(`Júlia tiene ${edades["Júlia"]}`);  
// → Júlia tiene 62  
console.log("¿Se conoce la edad de Jack?", "Jack" in edades);  
// → ¿Se conoce la edad de Jack? false  
console.log("¿Se conoce la edad de toString?", "toString" in  
edades);  
// → ¿Se conoce la edad de toString? true
```

Aquí, los nombres de propiedad del objeto son los nombres de las personas, y los valores de las propiedades son sus edades. Pero ciertamente no listamos a nadie con el nombre `toString` en nuestro mapa. Sin embargo, dado que los objetos simples derivan de `Object.prototype`, parece que la propiedad está allí.

Por lo tanto, usar objetos simples como mapas es peligroso. Hay varias formas posibles de evitar este problema. Primero, es posible crear objetos sin *ningún* prototipo. Si pasas `null` a `Object.create`, el objeto resultante no derivará de `Object.prototype` y se puede usar de forma segura como un mapa.

```
console.log("toString" in Object.create(null));  
// → false
```

Los nombres de las propiedades de los objetos deben ser cadenas. Si necesitas un mapa cuyas claves no puedan convertirse fácilmente en cadenas—como objetos—no puedes usar un objeto como tu mapa.

Afortunadamente, JavaScript viene con una clase llamada `Map` que está escrita para este propósito exacto. Almacena un mapeo y permite cualquier tipo de claves.

```
let ages = new Map();  
ages.set("Boris", 39);  
ages.set("Liang", 22);  
ages.set("Júlia", 62);  
  
console.log(`Júlia tiene ${ages.get("Júlia")}`);  
// → Júlia tiene 62  
console.log("¿Se conoce la edad de Jack?", ages.has("Jack"));  
// → ¿Se conoce la edad de Jack? false  
console.log(ages.has("toString"));  
// → false
```

Los métodos `set`, `get` y `has` forman parte de la interfaz del objeto `Map`. Escribir una estructura de datos que pueda actualizar y buscar rápidamente un gran conjunto de valores no es fácil, pero no tenemos que preocuparnos por eso. Alguien más lo hizo por

nosotros, y podemos utilizar su trabajo a través de esta interfaz sencilla.

Si tienes un objeto simple que necesitas tratar como un mapa por alguna razón, es útil saber que `Object.keys` devuelve solo las claves *propias* de un objeto, no las del prototipo. Como alternativa al operador `in`, puedes utilizar la función `Object.hasOwnProperty`, que ignora el prototipo del objeto.

```
console.log(Object.hasOwnProperty({x: 1}, "x"));  
// → true  
console.log(Object.hasOwnProperty({x: 1}, "toString"));  
// → false
```

POLIMORFISMO

Cuando llamas a la función `String` (que convierte un valor a una cadena) en un objeto, llamará al método `toString` en ese objeto para intentar crear una cadena significativa a partir de él. Mencione que algunos de los prototipos estándar definen su propia versión de `toString` para poder crear una cadena que contenga información más útil que `"[object Object]"`. También puedes hacerlo tú mismo.

```
Rabbit.prototype.toString = function() {  
  return `un conejo ${this.type}`;  
};  
  
console.log(String(conejoAsesino));  
// → un conejo asesino
```

Este es un ejemplo simple de una idea poderosa. Cuando se escribe un código para trabajar con objetos que tienen una determinada

interfaz, en este caso, un método `toString`, cualquier tipo de objeto que accidentalmente admita esta interfaz puede ser enchufado en el código, y este podrá funcionar con él.

Esta técnica se llama *polimorfismo*. El código polimórfico puede trabajar con valores de diferentes formas, siempre y cuando admitan la interfaz que espera.

Un ejemplo de una interfaz ampliamente utilizada es la de los objeto similar a un array que tiene una propiedad `length` que contiene un número, y propiedades numeradas para cada uno de sus elementos. Tanto los arreglos como las cadenas admiten esta interfaz, al igual que varios otros objetos, algunos de los cuales veremos más adelante en los capítulos sobre el navegador. Nuestra implementación de `forEach` en el [Capítulo 5](#) funciona en cualquier cosa que proporcione esta interfaz. De hecho, también lo hace `Array.prototype.forEach`.

```
Array.prototype.forEach.call({  
  length: 2,  
  0: "A",  
  1: "B"  
}, elt => console.log(elt));  
// → A  
// → B
```

GETTERS, SETTERS Y ESTÁTICOS

Las interfaces a menudo contienen propiedades simples, no solo métodos. Por ejemplo, los objetos `Map` tienen una propiedad `size` que te dice cuántas claves están almacenadas en ellos.

No es necesario que dicho objeto calcule y almacene directamente esa propiedad en la instancia. Incluso las propiedades que se acceden directamente pueden ocultar una llamada a un método. Dichos métodos se llaman *getter* y se definen escribiendo `get` delante del nombre del método en una expresión de objeto o declaración de clase.

```
let varyingSize = {  
  get size() {  
    return Math.floor(Math.random() * 100);  
  }  
};  
  
console.log(varyingSize.size);  
// → 73  
console.log(varyingSize.size);  
// → 49
```

Cada vez que alguien lee la propiedad `size` de este objeto, se llama al método asociado. Puedes hacer algo similar cuando se escribe en una propiedad, utilizando un *setter*.

```
class Temperature {  
  constructor(celsius) {  
    this.celsius = celsius;  
  }  
  get fahrenheit() {  
    return this.celsius * 1.8 + 32;  
  }  
  set fahrenheit(value) {  
    this.celsius = (value - 32) / 1.8;  
  }  
  
  static fromFahrenheit(value) {  
    return new Temperature((value - 32) / 1.8);  
  }  
}
```

```
let temp = new Temperature(22);
console.log(temp.fahrenheit);
// → 71.6
temp.fahrenheit = 86;
console.log(temp.celsius);
// → 30
```

La clase `Temperature` te permite leer y escribir la temperatura en grados Celsius o grados Fahrenheit, pero internamente solo almacena Celsius y convierte automáticamente de y a Celsius en el *getter* y *setter* de `fahrenheit`.

A veces quieres adjuntar algunas propiedades directamente a tu función constructora, en lugar de al prototipo. Estos métodos no tendrán acceso a una instancia de clase, pero pueden, por ejemplo, usarse para proporcionar formas adicionales de crear instancias.

Dentro de una declaración de clase, los métodos o propiedades que tienen `static` escrito antes de su nombre se almacenan en el constructor. Por lo tanto, la clase `Temperature` te permite escribir `Temperature.fromFahrenheit(100)` para crear una temperatura usando grados Fahrenheit.

SÍMBOLOS

Mencioné en [Capítulo 4](#), que un bucle `for/of` puede recorrer varios tipos de estructuras de datos. Este es otro caso de polimorfismo: tales bucles esperan que la estructura de datos exponga una interfaz específica, la cual hacen los arrays y las cadenas. ¡Y también podemos agregar esta interfaz a nuestros propios objetos! Pero antes de hacerlo, debemos echar un vistazo breve al tipo de símbolo.

Es posible que múltiples interfaces utilicen el mismo nombre de propiedad para diferentes cosas. Por ejemplo, en objetos similares a arrays, `length` se refiere a la cantidad de elementos en la colección. Pero una interfaz de objeto que describa una ruta de senderismo podría usar `length` para proporcionar la longitud de la ruta en metros. No sería posible que un objeto cumpla con ambas interfaces.

Un objeto que intente ser una ruta y similar a un array (quizás para enumerar sus puntos de referencia) es algo un tanto improbable, y este tipo de problema no es tan común en la práctica. Pero para cosas como el protocolo de iteración, los diseñadores del lenguaje necesitaban un tipo de propiedad que *realmente* no entrara en conflicto con ninguna otra. Por lo tanto, en 2015, se agregaron los *símbolos* al lenguaje.

La mayoría de las propiedades, incluidas todas las propiedades que hemos visto hasta ahora, se nombran con cadenas. Pero también es posible usar símbolos como nombres de propiedades. Los símbolos son valores creados con la función `Symbol`. A diferencia de las cadenas, los símbolos recién creados son únicos: no puedes crear el mismo símbolo dos veces.

```
let sym = Symbol("nombre");
console.log(sym == Symbol("nombre"));
// → false
Rabbit.prototype[sym] = 55;
console.log(killerRabbit[sym]);
// → 55
```

La cadena que pasas a `Symbol` se incluye cuando la conviertes en una cadena y puede facilitar reconocer un símbolo cuando, por

ejemplo, se muestra en la consola. Pero no tiene otro significado más allá de eso: varios símbolos pueden tener el mismo nombre.

Ser tanto únicos como utilizables como nombres de propiedades hace que los símbolos sean adecuados para definir interfaces que pueden convivir pacíficamente junto a otras propiedades, independientemente de cuáles sean sus nombres.

```
const longitud = Symbol("longitud");
Array.prototype[longitud] = 0;

console.log([1, 2].length);
// → 2
console.log([1, 2][longitud]);
// → 0
```

Es posible incluir propiedades de símbolos en expresiones de objetos y clases mediante el uso de corchetes. Esto hace que la expresión entre los corchetes se evalúe para producir el nombre de la propiedad, análogo a la notación de acceso a propiedades mediante corchetes cuadrados.

```
let miViaje = {
  longitud: 2,
  0: "Lankwitz",
  1: "Babelsberg",
  [longitud]: 21500
};
console.log(miViaje[longitud], miViaje.longitud);
// → 21500 2
```

LA INTERFAZ DEL ITERADOR

Se espera que el objeto proporcionado a un bucle `for/of` sea *iterable*. Esto significa que tiene un método nombrado con el

símbolo `Symbol.iterator` (un valor de símbolo definido por el lenguaje, almacenado como una propiedad de la función `Symbol`).

Cuando se llama, ese método debería devolver un objeto que proporcione una segunda interfaz, *iterador*. Este es lo que realmente itera. Tiene un método `next` que devuelve el próximo resultado. Ese resultado debería ser un objeto con una propiedad `value` que proporciona el siguiente valor, si lo hay, y una propiedad `done`, que debería ser `true` cuando no hay más resultados y `false` en caso contrario.

Ten en cuenta que los nombres de propiedad `next`, `value` y `done` son simples cadenas, no símbolos. Solo `Symbol.iterator`, que probablemente se agregará a *muchos* objetos diferentes, es un símbolo real.

Podemos usar esta interfaz directamente nosotros mismos.

```
let okIterador = "OK"[Symbol.iterator]();
console.log(okIterador.next());
// → {value: "O", done: false}
console.log(okIterador.next());
// → {value: "K", done: false}
console.log(okIterador.next());
// → {value: undefined, done: true}
```

Implementemos una estructura de datos iterable similar a la lista enlazada del ejercicio en [Capítulo ?](#). Esta vez escribiremos la lista como una clase.

```
class List {
  constructor(value, rest) {
    this.value = value;
    this.rest = rest;
  }
}
```

```

    }

    get length() {
        return 1 + (this.rest ? this.rest.length : 0);
    }

    static fromArray(array) {
        let result = null;
        for (let i = array.length - 1; i >= 0; i--) {
            result = new this(array[i], result);
        }
        return result;
    }
}

```

Toma en cuenta que `this`, en un método estático, apunta al constructor de la clase, no a una instancia, ya que no hay una instancia disponible cuando se llama a un método estático.

Iterar sobre una lista debería devolver todos los elementos de la lista desde el principio hasta el final. Escribiremos una clase separada para el iterador.

```

class ListIterator {
    constructor(list) {
        this.list = list;
    }

    next() {
        if (this.list == null) {
            return { done: true };
        }
        let value = this.list.value;
        this.list = this.list.rest;
        return { value, done: false };
    }
}

```

La clase realiza un seguimiento del progreso de la iteración a través de la lista actualizando su propiedad `list` para moverse al siguiente objeto de lista cada vez que se devuelve un valor, y reporta que ha terminado cuando esa lista está vacía (`null`).

Ahora configuraremos la clase `List` para que sea iterable. A lo largo de este libro, ocasionalmente utilizaré la manipulación de prototipos posterior al hecho para agregar métodos a las clases de modo que las piezas individuales de código se mantengan pequeñas y autónomas. En un programa regular, donde no hay necesidad de dividir el código en piezas pequeñas, declararías estos métodos directamente en la clase en su lugar.

```
List.prototype[Symbol.iterator] = function() {  
  return new ListIterator(this);  
};
```

Ahora podemos iterar sobre una lista con `for/of`.

```
let lista = List.fromArray([1, 2, 3]);  
for (let elemento of lista) {  
  console.log(elemento);  
}  
// → 1  
// → 2  
// → 3
```

La sintaxis `...` en notación de arrays y en llamadas a funciones funciona de forma similar con cualquier objeto iterable. Por ejemplo, puedes usar `[...valor]` para crear un array que contenga los elementos de un objeto iterable arbitrario.

```
console.log([... "PCI"]);  
// → ["P", "C", "I"]
```

HERENCIA

Imaginemos que necesitamos un tipo de lista, bastante parecido a la clase `List` que vimos anteriormente, pero como siempre estaremos preguntando por su longitud, no queremos tener que recorrer su `rest` cada vez, en su lugar, queremos almacenar la longitud en cada instancia para un acceso eficiente.

El sistema de prototipos de JavaScript permite crear una *nueva* clase, muy similar a la clase antigua, pero con nuevas definiciones para algunas de sus propiedades. El prototipo de la nueva clase se deriva del prototipo antiguo pero agrega una nueva definición, por ejemplo, para el `getter` de `length`.

En términos de programación orientada a objetos, esto se llama *herencia*. La nueva clase hereda propiedades y comportamientos de la clase antigua.

```
class LengthList extends List {
  #length;

  constructor(valor, rest) {
    super(valor, rest);
    this.#length = super.length;
  }

  get length() {
    return this.#length;
  }
}

console.log(LengthList.fromArray([1, 2, 3]).length);
// → 3
```

El uso de la palabra `extends` indica que esta clase no debería basarse directamente en el prototipo predeterminado de `Object`, sino en alguna otra clase. Esta se llama la *superclase*. La clase derivada es la *subclase*.

Para inicializar una instancia de `LengthList`, el constructor llama al constructor de su superclase a través de la palabra clave `super`. Esto es necesario porque si este nuevo objeto se va a comportar (aproximadamente) como una `List`, va a necesitar las propiedades de instancia que tienen las listas.

Luego, el constructor almacena la longitud de la lista en una propiedad privada. Si hubiéramos escrito `this.longitud` ahí, se habría llamado al getter de la propia clase, lo cual no funciona aún, ya que `#longitud` aún no ha sido completado. Podemos usar `super.algo` para llamar a métodos y getters en el prototipo de la superclase, lo cual a menudo es útil.

La herencia nos permite construir tipos de datos ligeramente diferentes a partir de tipos de datos existentes con relativamente poco trabajo. Es una parte fundamental de la tradición orientada a objetos, junto con la encapsulación y la polimorfismo. Pero, mientras que los dos últimos se consideran generalmente ideas maravillosas, la herencia es más controvertida.

Mientras que encapsulación y polimorfismo se pueden utilizar para *separar* las piezas de código unas de otras, reduciendo el enredo del programa en general, herencia fundamentalmente ata clases juntas, creando *más* enredo. Al heredar de una clase, generalmente tienes que saber más sobre cómo funciona que cuando simplemente la usas. La herencia puede ser una herramienta útil para hacer que algunos

tipos de programas sean más concisos, pero no debería ser la primera herramienta a la que recurras, y probablemente no deberías buscar activamente oportunidades para construir jerarquías de clases (árboles genealógicos de clases).

EL OPERADOR INSTANCEOF

A veces es útil saber si un objeto se derivó de una clase específica. Para esto, JavaScript proporciona un operador binario llamado `instanceof`.

```
console.log(
  new LengthList(1, null) instanceof LengthList);
// → true
console.log(new LengthList(2, null) instanceof List);
// → true
console.log(new List(3, null) instanceof LengthList);
// → false
console.log([1] instanceof Array);
// → true
```

El operador podrá ver a través de tipos heredados, por lo que un `LengthList` es una instancia de `List`. El operador también se puede aplicar a constructores estándar como `Array`. Casi todo objeto es una instancia de `Object`.

RESUMEN

Los objetos hacen más que simplemente contener sus propias propiedades. Tienen prototipos, que son otros objetos. Actuarán como si tuvieran propiedades que no tienen siempre y cuando su prototipo tenga esa propiedad. Los objetos simples tienen `Object.prototype` como su prototipo.

Los constructores, que son funciones cuyos nombres generalmente comienzan con una letra mayúscula, se pueden usar con el operador `new` para crear nuevos objetos. El prototipo del nuevo objeto será el objeto encontrado en la propiedad `prototype` del constructor. Puedes sacar buen provecho de esto poniendo las propiedades que comparten todos los valores de un tipo dado en su prototipo. Existe una notación de `class` que proporciona una forma clara de definir un constructor y su prototipo.

Puedes definir getters y setters para llamar secretamente a métodos cada vez que se accede a una propiedad de un objeto. Los métodos estáticos son métodos almacenados en el constructor de una clase, en lugar de en su prototipo.

El operador `instanceof` puede, dado un objeto y un constructor, decirte si ese objeto es una instancia de ese constructor.

Una cosa útil que se puede hacer con objetos es especificar una interfaz para ellos y decirle a todo el mundo que se supone que deben comunicarse con tu objeto solo a través de esa interfaz. El resto de los detalles que componen tu objeto están ahora *encapsulados*, escondidos detrás de la interfaz. Puedes usar propiedades privadas para ocultar una parte de tu objeto del mundo exterior.

Más de un tipo puede implementar la misma interfaz. El código escrito para usar una interfaz automáticamente sabe cómo trabajar con cualquier número de objetos diferentes que proporcionen la interfaz. Esto se llama *polimorfismo*.

Cuando se implementan múltiples clases que difieren solo en algunos detalles, puede ser útil escribir las nuevas clases como

subclases de una clase existente, *heredando* parte de su comportamiento.

EJERCICIOS

UN TIPO DE VECTOR

Escribe una clase `Vec` que represente un vector en el espacio bidimensional. Toma los parámetros `x` e `y` (números), que debería guardar en propiedades del mismo nombre.

Dale a la clase `Vec` dos métodos en su prototipo, `plus` y `minus`, que tomen otro vector como parámetro y devuelvan un nuevo vector que tenga la suma o la diferencia de los valores `x` e `y` de los dos vectores (`this` y el parámetro).

Agrega una propiedad getter `length` al prototipo que calcule la longitud del vector, es decir, la distancia del punto (x, y) desde el origen $(0, 0)$.

GRUPOS

El entorno estándar de JavaScript proporciona otra estructura de datos llamada `Set`. Al igual que una instancia de `Map`, un conjunto contiene una colección de valores. A diferencia de `Map`, no asocia otros valores con esos, solo realiza un seguimiento de qué valores forman parte del conjunto. Un valor puede formar parte de un conjunto solo una vez: agregarlo nuevamente no tiene ningún efecto.

Escribe una clase llamada `Group` (ya que `Set` está siendo utilizado). Al igual que `Set`, tiene los métodos `add`, `delete` y `has`. Su constructor crea un grupo vacío, `add` agrega un valor al grupo (pero

solo si aún no es miembro), `delete` elimina su argumento del grupo (si era miembro), y `has` devuelve un valor booleano que indica si su argumento es miembro del grupo.

Usa el operador `===`, o algo equivalente como `indexOf`, para determinar si dos valores son iguales.

Dale a la clase un método estático `from` que tome un objeto iterable como argumento y cree un grupo que contenga todos los valores producidos al iterar sobre él.

GRUPOS ITERABLES

Haz que la clase `Group` del ejercicio anterior sea iterable. Refiérete a la sección sobre la interfaz del iterador anteriormente en el capítulo si no tienes claro la forma exacta de la interfaz.

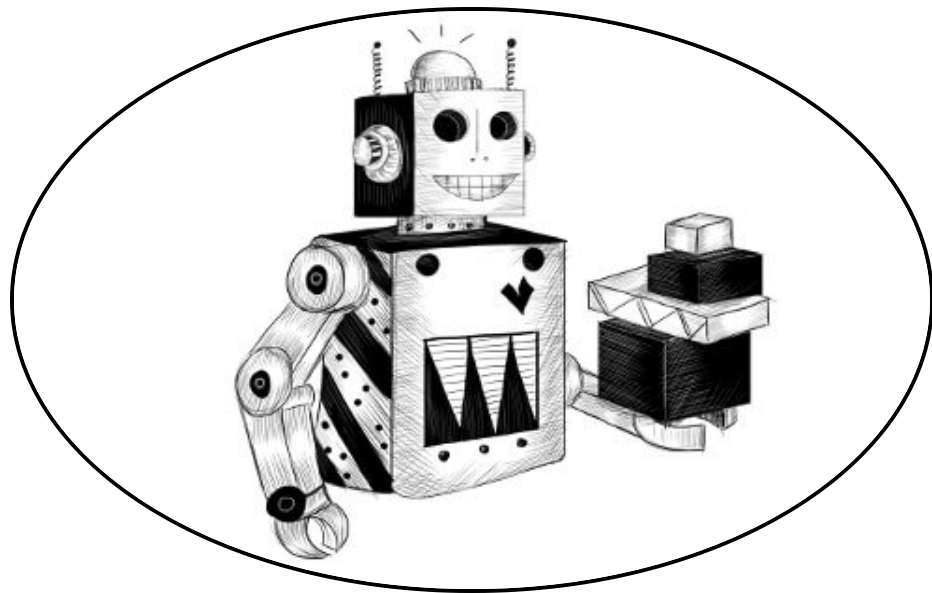
Si utilizaste un array para representar los miembros del grupo, no devuelvas simplemente el iterador creado al llamar al método `Symbol.iterator` en el array. Eso funcionaría, pero va en contra del propósito de este ejercicio.

Está bien si tu iterador se comporta de manera extraña cuando el grupo se modifica durante la iteración.

PROYECTO: UN ROBOT

“[...] la pregunta de si las Máquinas Pueden Pensar [...] es tan relevante como la pregunta de si los Submarinos Pueden Nadar.”

—Edsger Dijkstra, *Las amenazas a la ciencia informática*



En los capítulos del “proyecto”, dejaré de golpearte con nueva teoría por un breve momento, y en su lugar trabajaremos en un programa juntos. La teoría es necesaria para aprender a programar, pero leer y entender programas reales es igual de importante.

Nuestro proyecto en este capítulo es construir un autómeta, un pequeño programa que realiza una tarea en un mundo virtual. Nuestro autómeta será un robot de entrega de correo que recoge y deja paquetes.

MEADOWFIELD

El pueblo de Meadowfield no es muy grande. Consiste en 11 lugares con 14 carreteras entre ellos. Se puede describir con este array de carreteras:

```
const roads = [  
  "Alice's House-Bob's House",    "Alice's House-Cabin",  
  "Alice's House-Post Office",     "Bob's House-Town Hall",  
  "Daria's House-Ernie's House",   "Daria's House-Town Hall",  
  "Ernie's House-Grete's House",   "Grete's House-Farm",  
  "Grete's House-Shop",            "Marketplace-Farm",  
  "Marketplace-Post Office",       "Marketplace-Shop",  
  "Marketplace-Town Hall",         "Shop-Town Hall"  
];
```



La red de carreteras en el pueblo forma un *gráfico*. Un gráfico es una colección de puntos (lugares en el pueblo) con líneas entre ellos (carreteras). Este gráfico será el mundo por el que se moverá nuestro robot.

El array de cadenas no es muy fácil de trabajar. Lo que nos interesa son los destinos a los que podemos llegar desde un lugar dado. Vamos a convertir la lista de carreteras en una estructura de datos que, para cada lugar, nos diga qué se puede alcanzar desde allí.

```
function buildGraph(edges) {
  let graph = Object.create(null);
  function addEdge(from, to) {
    if (from in graph) {
      graph[from].push(to);
    } else {
      graph[from] = [to];
    }
  }
  for (let [from, to] of edges.map(r => r.split("-"))) {
    addEdge(from, to);
    addEdge(to, from);
  }
  return graph;
}

const roadGraph = buildGraph(roads);
```

Dado un array de aristas, `buildGraph` crea un objeto de mapa que, para cada nodo, almacena un array de nodos conectados.

Utiliza el método `split` para pasar de las cadenas de carreteras, que tienen la forma "Inicio-Fin", a arrays de dos elementos que contienen el inicio y el fin como cadenas separadas.

LA TAREA

Nuestro robot se moverá por el pueblo. Hay paquetes en varios lugares, cada uno dirigido a algún otro lugar. El robot recoge los paquetes cuando llega a ellos y los entrega cuando llega a sus destinos.

El autómata debe decidir, en cada punto, hacia dónde ir a continuación. Habrá terminado su tarea cuando todos los paquetes hayan sido entregados.

Para poder simular este proceso, debemos definir un mundo virtual que pueda describirlo. Este modelo nos dice dónde está el robot y dónde están los paquetes. Cuando el robot decide moverse a algún lugar, necesitamos actualizar el modelo para reflejar la nueva situación.

Si estás pensando en términos de programación orientada a objetos, tu primer impulso podría ser empezar a definir objetos para los diferentes elementos en el mundo: una clase para el robot, una para un paquete, tal vez una para lugares. Estos podrían tener propiedades que describen su estado actual, como la pila de paquetes en un lugar, que podríamos cambiar al actualizar el mundo.

Esto es incorrecto. Al menos, usualmente lo es. El hecho de que algo suene como un objeto no significa automáticamente que deba ser un objeto en tu programa. Escribir reflexivamente clases para cada concepto en tu aplicación tiende a dejarte con una colección de objetos interconectados que tienen su propio estado interno cambiante. Estos programas a menudo son difíciles de entender y, por lo tanto, fáciles de romper.

En lugar de eso, vamos a condensar el estado del pueblo en el conjunto mínimo de valores que lo define. Está la ubicación actual del robot y la colección de paquetes no entregados, cada uno de los cuales tiene una ubicación actual y una dirección de destino. Eso es todo.

Y mientras lo hacemos, hagamos que no *cambiamos* este estado cuando el robot se mueve, sino que calculemos un *nuevo* estado para la situación después del movimiento.

```
class VillageState {
  constructor(place, parcels) {
    this.place = place;
    this.parcels = parcels;
  }

  move(destination) {
    if (!roadGraph[this.place].includes(destination)) {
      return this;
    } else {
      let parcels = this.parcels.map(p => {
        if (p.place !== this.place) return p;
        return {place: destination, address: p.address};
      }).filter(p => p.place !== p.address);
      return new VillageState(destination, parcels);
    }
  }
}
```

El método `move` es donde ocurre la acción. Primero verifica si hay un camino desde el lugar actual hasta el destino, y si no lo hay, devuelve el estado anterior ya que este no es un movimiento válido.

Luego crea un nuevo estado con el destino como el nuevo lugar del robot. Pero también necesita crear un nuevo conjunto de paquetes: los paquetes que lleva el robot (que están en el lugar actual del robot) deben ser trasladados al nuevo lugar. Y los paquetes dirigidos al nuevo lugar deben ser entregados, es decir, deben ser eliminados del conjunto de paquetes no entregados. La llamada a `map` se encarga del traslado y la llamada a `filter` de la entrega.

Los objetos de parcela no se modifican cuando se mueven, sino que se vuelven a crear. El método `move` nos proporciona un nuevo estado de aldea pero deja intacto por completo el anterior.

```
let first = new VillageState(
  "Oficina de Correos",
  [{place: "Oficina de Correos", address: "Casa de Alice"}]
);
let next = first.move("Casa de Alice");

console.log(next.place);
// → Casa de Alice
console.log(next.parcels);
// → []
console.log(first.place);
// → Oficina de Correos
```

El movimiento hace que la parcela se entregue, y esto se refleja en el siguiente estado. Pero el estado inicial sigue describiendo la situación en la que el robot está en la oficina de correos y la parcela no se ha entregado.

DATOS PERSISTENTES

Las estructuras de datos que no cambian se llaman *inmutables* o *persistentes*. Se comportan de manera similar a las cadenas de texto y los números en el sentido de que son lo que son y se mantienen así, en lugar de contener cosas diferentes en momentos diferentes.

En JavaScript, casi todo *puede* cambiarse, por lo que trabajar con valores que se supone que son persistentes requiere cierta moderación. Existe una función llamada `Object.freeze` que cambia un objeto para que la escritura en sus propiedades sea ignorada. Podrías usar esto para asegurarte de que tus objetos no se

modifiquen, si así lo deseas. Congelar requiere que la computadora realice un trabajo adicional, y que las actualizaciones se ignoren es casi tan propenso a confundir a alguien como hacer que hagan lo incorrecto. Por lo tanto, suelo preferir simplemente decirle a las personas que un objeto dado no debe ser modificado y esperar que lo recuerden.

```
let object = Object.freeze({value: 5});  
object.value = 10;  
console.log(object.value);  
// → 5
```

¿Por qué me estoy esforzando tanto en no cambiar los objetos cuando el lenguaje obviamente espera que lo haga?

Porque me ayuda a entender mis programas. Una vez más, esto se trata de gestionar la complejidad. Cuando los objetos en mi sistema son cosas fijas y estables, puedo considerar operaciones sobre ellos de forma aislada: moverse a la casa de Alice desde un estado inicial dado siempre produce el mismo nuevo estado. Cuando los objetos cambian con el tiempo, eso añade toda una nueva dimensión de complejidad a este tipo de razonamiento.

Para un sistema pequeño como el que estamos construyendo en este capítulo, podríamos manejar ese poco de complejidad extra. Pero el límite más importante respecto a qué tipo de sistemas podemos construir es cuánto podemos entender. Cualquier cosa que haga que tu código sea más fácil de entender te permite construir un sistema más ambicioso.

Desafortunadamente, aunque entender un sistema construido sobre estructuras de datos persistentes es más fácil, *diseñar* uno,

especialmente cuando tu lenguaje de programación no ayuda, puede ser un poco más difícil. Buscaremos oportunidades para usar estructuras de datos persistentes en este libro, pero también usaremos aquellas que pueden cambiar.

SIMULACIÓN

Un robot de entrega observa el mundo y decide en qué dirección quiere moverse. Como tal, podríamos decir que un robot es una función que toma un objeto `VillageState` y devuelve el nombre de un lugar cercano.

Dado que queremos que los robots puedan recordar cosas, para que puedan hacer y ejecutar planes, también les pasamos su memoria y les permitimos devolver una nueva memoria. Por lo tanto, lo que un robot devuelve es un objeto que contiene tanto la dirección en la que quiere moverse como un valor de memoria que se le dará la próxima vez que se llame.

```
function runRobot(state, robot, memory) {
  for (let turn = 0;; turn++) {
    if (state.parcels.length == 0) {
      console.log(`Terminado en ${turn} turnos`);
      break;
    }
    let action = robot(state, memory);
    state = state.move(action.direction);
    memory = action.memory;
    console.log(`Movido a ${action.direction}`);
  }
}
```

Consideremos lo que un robot tiene que hacer para “resolver” un estado dado. Debe recoger todos los paquetes visitando cada

ubicación que tenga un paquete y entregarlos visitando cada ubicación a la que esté dirigido un paquete, pero solo después de recoger el paquete.

¿Cuál es la estrategia más tonta que podría funcionar? El robot podría simplemente caminar en una dirección aleatoria en cada turno. Eso significa que, con gran probabilidad, eventualmente se topará con todos los paquetes y en algún momento también llegará al lugar donde deben ser entregados.

Esto es cómo podría lucir eso:

```
function randomPick(array) {  
  let choice = Math.floor(Math.random() * array.length);  
  return array[choice];  
}  
  
function randomRobot(state) {  
  return {direction: randomPick(roadGraph[state.place])};  
}
```

Recuerda que `Math.random()` devuelve un número entre cero y uno, pero siempre por debajo de uno. Multiplicar dicho número por la longitud de un array y luego aplicarle `Math.floor` nos da un índice aleatorio para el array.

Dado que este robot no necesita recordar nada, ignora su segundo argumento (recuerda que las funciones de JavaScript pueden ser llamadas con argumentos adicionales sin efectos adversos) y omite la propiedad `memory` en su objeto devuelto.

Para poner a trabajar a este sofisticado robot, primero necesitaremos una forma de crear un nuevo estado con algunos paquetes. Un

método estático (escrito aquí añadiendo directamente una propiedad al constructor) es un buen lugar para poner esa funcionalidad.

```
VillageState.random = function(parcelCount = 5) {  
  let parcels = [];  
  for (let i = 0; i < parcelCount; i++) {  
    let address = randomPick(Object.keys(roadGraph));  
    let place;  
    do {  
      place = randomPick(Object.keys(roadGraph));  
    } while (place == address);  
    parcels.push({place, address});  
  }  
  return new VillageState("Oficina Postal", parcels);  
};
```

No queremos ningún paquete que sea enviado desde el mismo lugar al que está dirigido. Por esta razón, el bucle `do` sigue eligiendo nuevos lugares cuando obtiene uno que es igual a la dirección.

Vamos a iniciar un mundo virtual.

```
runRobot(VillageState.random(), randomRobot);  
// → Movido a Mercado  
// → Movido a Ayuntamiento  
// → ...  
// → Terminado en 63 turnos
```

Al robot le lleva muchas vueltas entregar los paquetes porque no está planificando muy bien. Abordaremos eso pronto.

ruta del camión de correo

Deberíamos poder hacerlo mucho mejor que el robot aleatorio. Una mejora sencilla sería inspirarnos en la forma en que funciona la entrega de correo en el mundo real. Si encontramos una ruta que

pase por todos los lugares del pueblo, el robot podría recorrer esa ruta dos veces, momento en que se garantizaría que ha terminado. Aquí tienes una de esas rutas (comenzando desde la oficina de correos):

```
const mailRoute = [  
  "Casa de Alicia", "Cabaña", "Casa de Alicia", "Casa de Bob",  
  "Ayuntamiento", "Casa de Daria", "Casa de Ernie",  
  "Casa de Grete", "Tienda", "Casa de Grete", "Granja",  
  "Plaza del Mercado", "Oficina de Correos"  
];
```

Para implementar el robot que sigue la ruta, necesitaremos hacer uso de la memoria del robot. El robot guarda el resto de su ruta en su memoria y deja caer el primer elemento en cada turno.

```
function routeRobot(state, memory) {  
  if (memory.length == 0) {  
    memory = mailRoute;  
  }  
  return {direction: memory[0], memory: memory.slice(1)};  
}
```

Este robot es mucho más rápido ya. Tomará un máximo de 26 vueltas (el doble de la ruta de 13 pasos) pero generalmente menos.

BÚSQUEDA DE CAMINOS

Aún así, no llamaría a seguir ciegamente una ruta fija un comportamiento inteligente. Sería más eficiente si el robot ajustara su comportamiento a la tarea real que debe realizarse.

Para hacer eso, tiene que poder moverse deliberadamente hacia un paquete dado o hacia la ubicación donde se debe entregar un

paquete. Hacer eso, incluso cuando el objetivo está a más de un movimiento de distancia, requerirá algún tipo de función de búsqueda de ruta.

El problema de encontrar una ruta a través de un grafo es un *problema de búsqueda* típico. Podemos determinar si una solución dada (una ruta) es una solución válida, pero no podemos calcular directamente la solución como podríamos hacerlo para $2 + 2$. En su lugar, debemos seguir creando soluciones potenciales hasta encontrar una que funcione.

El número de rutas posibles a través de un grafo es infinito. Pero al buscar una ruta de A a B , solo estamos interesados en aquellas que comienzan en A . Además, no nos importan las rutas que visiten el mismo lugar dos veces, esas definitivamente no son las rutas más eficientes en ningún lugar. Así que eso reduce la cantidad de rutas que el buscador de rutas debe considerar. De hecho, estamos mayormente interesados en la ruta *más corta*. Por lo tanto, queremos asegurarnos de buscar rutas cortas antes de mirar las más largas. Un buen enfoque sería “expandir” rutas desde el punto de inicio, explorando cada lugar alcanzable que aún no haya sido visitado, hasta que una ruta llegue al objetivo. De esta manera, solo exploraremos rutas que sean potencialmente interesantes, y sabremos que la primera ruta que encontremos es la ruta más corta (o una de las rutas más cortas, si hay más de una).

Aquí hay una función que hace esto:

```
function findRoute(graph, from, to) {  
  let work = [{at: from, route: []}];  
  for (let i = 0; i < work.length; i++) {  
    let {at, route} = work[i];
```

```

    for (let place of graph[at]) {
      if (place == to) return route.concat(place);
      if (!work.some(w => w.at == place)) {
        work.push({at: place, route: route.concat(place)});
      }
    }
  }
}

```

La exploración debe realizarse en el orden correcto: los lugares que se alcanzaron primero deben explorarse primero. No podemos explorar de inmediato un lugar tan pronto como lleguemos a él porque eso significaría que los lugares alcanzados *desde allí* también se explorarían de inmediato, y así sucesivamente, incluso si puede haber otros caminos más cortos que aún no se han explorado.

Por lo tanto, la función mantiene una *lista de trabajo*. Esta es una matriz de lugares que deben ser explorados a continuación, junto con la ruta que nos llevó allí. Comienza con solo la posición de inicio y una ruta vacía.

La búsqueda luego opera tomando el siguiente elemento en la lista y explorándolo, lo que significa que se ven todas las rutas que salen de ese lugar. Si una de ellas es el objetivo, se puede devolver una ruta terminada. De lo contrario, si no hemos mirado este lugar antes, se agrega un nuevo elemento a la lista. Si lo hemos mirado antes, dado que estamos buscando rutas cortas primero, hemos encontrado o bien una ruta más larga a ese lugar o una exactamente tan larga como la existente, y no necesitamos explorarla.

Puedes imaginar visualmente esto como una red de rutas conocidas que se extienden desde la ubicación de inicio, creciendo de manera uniforme en todos los lados (pero nunca enredándose de nuevo en sí

misma). Tan pronto como el primer hilo alcance la ubicación objetivo, ese hilo se rastrea de vuelta al inicio, dándonos nuestra ruta.

Nuestro código no maneja la situación en la que no hay más elementos de trabajo en la lista de trabajo porque sabemos que nuestro gráfico está *conectado*, lo que significa que se puede llegar a cada ubicación desde todas las demás ubicaciones. Siempre podremos encontrar una ruta entre dos puntos, y la búsqueda no puede fallar.

```
function goalOrientedRobot({place, parcels}, route) {
  if (route.length == 0) {
    let parcel = parcels[0];
    if (parcel.place != place) {
      route = findRoute(roadGraph, place, parcel.place);
    } else {
      route = findRoute(roadGraph, place, parcel.address);
    }
  }
  return {direction: route[0], memory: route.slice(1)};
}
```

Este robot utiliza el valor de su memoria como una lista de direcciones en las que moverse, al igual que el robot que sigue la ruta. Cuando esa lista está vacía, debe averiguar qué hacer a continuación. Toma el primer paquete no entregado del conjunto y, si ese paquete aún no ha sido recogido, traza una ruta hacia él. Si el paquete ya ha sido recogido, todavía necesita ser entregado, por lo que el robot crea una ruta hacia la dirección de entrega.

Este robot suele terminar la tarea de entregar 5 paquetes en aproximadamente 16 turnos. Eso es ligeramente mejor que `routeRobot` pero definitivamente no es óptimo.

EJERCICIOS

MEDICIÓN DE UN ROBOT

Es difícil comparar de manera objetiva los robots solo dejando que resuelvan algunos escenarios. Tal vez un robot simplemente tuvo tareas más fáciles o el tipo de tareas en las que es bueno, mientras que el otro no.

Escribe una función `compareRobots` que tome dos robots (y su memoria inicial). Debería generar 100 tareas y permitir que cada uno de los robots resuelva cada una de estas tareas. Cuando termine, debería mostrar el número promedio de pasos que cada robot dio por tarea.

Por el bien de la equidad, asegúrate de darle a cada tarea a ambos robots, en lugar de generar tareas diferentes por robot.

EFICIENCIA DEL ROBOT

¿Puedes escribir un robot que termine la tarea de entrega más rápido que `goalOrientedRobot`? Si observas el comportamiento de ese robot, ¿qué cosas claramente absurdas hace? ¿Cómo podrían mejorarse?

Si resolviste el ejercicio anterior, es posible que desees utilizar tu función `compareRobots` para verificar si mejoraste el robot.

GRUPO PERSISTENTE

La mayoría de las estructuras de datos proporcionadas en un entorno estándar de JavaScript no son muy adecuadas para un uso

persistente. Los Arrays tienen métodos `slice` y `concat`, que nos permiten crear fácilmente nuevos arrays sin dañar el antiguo. Pero `Set`, por ejemplo, no tiene métodos para crear un nuevo conjunto con un elemento añadido o eliminado.

Escribe una nueva clase `PGroup`, similar a la clase `Grupo` del [Capítulo 6](#), que almacena un conjunto de valores. Al igual que `Grupo`, tiene métodos `add`, `delete`, y `has`.

Sin embargo, su método `add` debería devolver una *nueva* instancia de `PGroup` con el miembro dado añadido y dejar la anterior sin cambios. De manera similar, `delete` crea una nueva instancia sin un miembro dado.

La clase debería funcionar para valores de cualquier tipo, no solo para strings. No tiene que ser eficiente cuando se utiliza con grandes cantidades de valores.

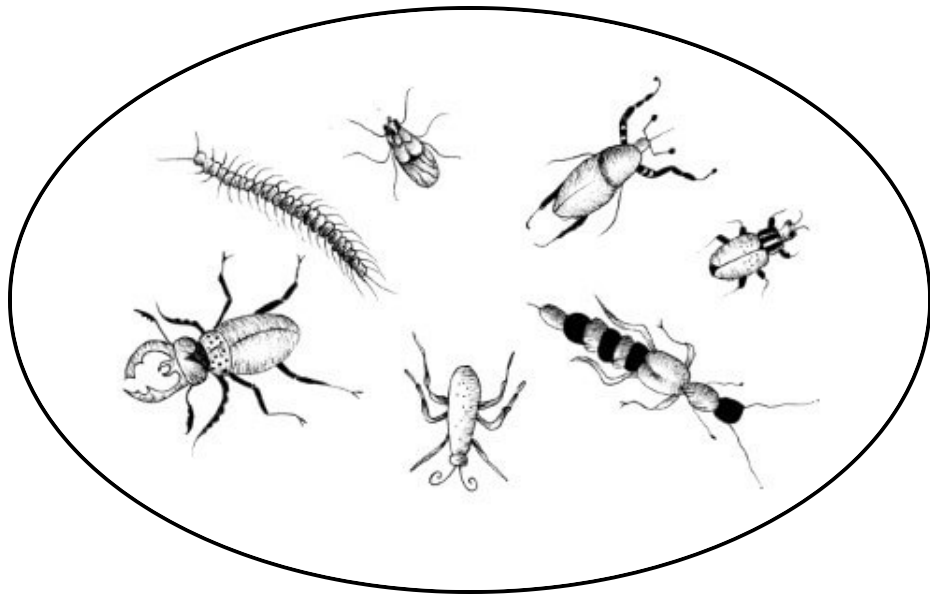
El constructor no debería ser parte de la interfaz de la clase (aunque definitivamente querrás usarlo internamente). En su lugar, hay una instancia vacía, `PGroup.empty`, que se puede usar como valor inicial.

¿Por qué necesitas solo un valor `PGroup.empty`, en lugar de tener una función que cree un nuevo mapa vacío cada vez?

BUGS Y ERRORES

“Depurar es el doble de difícil que escribir el código en primer lugar. Por lo tanto, si escribes el código lo más ingeniosamente posible, por definición, no eres lo suficientemente inteligente como para depurarlo.”

—Brian Kernighan and P.J. Plauger, *The Elements of Programming Style*



Las fallas en los programas de computadora generalmente se llaman *bugs*. Hace que los programadores se sientan bien imaginarlos como pequeñas cosas que simplemente se meten en nuestro trabajo. En realidad, por supuesto, nosotros mismos los colocamos allí.

Si un programa es pensamiento cristalizado, puedes clasificar aproximadamente los errores en aquellos causados por pensamientos confusos y aquellos causados por errores introducidos

al convertir un pensamiento en código. El primer tipo generalmente es más difícil de diagnosticar y arreglar que el último.

LENGUAJE

Muchos errores podrían ser señalados automáticamente por la computadora, si supiera lo suficiente sobre lo que estamos intentando hacer. Pero la laxitud de JavaScript es un obstáculo aquí. Su concepto de enlaces y propiedades es lo suficientemente vago como para rara vez atrapar typos antes de ejecutar realmente el programa. E incluso entonces, te permite hacer algunas cosas claramente absurdas sin quejarse, como calcular `true * "monkey"`.

Hay algunas cosas sobre las que JavaScript sí se queja. Escribir un programa que no siga la gramática del lenguaje hará que la computadora se queje de inmediato. Otras cosas, como llamar a algo que no es una función o buscar una propiedad en un valor `undefined` harán que se reporte un error cuando el programa intente realizar la acción.

Pero a menudo, tu cálculo absurdo simplemente producirá `NaN` (no es un número) o un valor indefinido, mientras que el programa continúa felizmente, convencido de que está haciendo algo significativo. El error se manifestará solo más tarde, después de que el valor falso haya pasado por varias funciones. Es posible que no desencadene un error en absoluto, pero silenciosamente cause que la salida del programa sea incorrecta. Encontrar la fuente de tales problemas puede ser difícil.

El proceso de encontrar errores—bugs—en los programas se llama *depuración*.

MODO ESTRICTO

JavaScript puede ser un *poco* más estricto al habilitar el *modo estricto*. Esto se hace colocando la cadena `"use strict"` en la parte superior de un archivo o en el cuerpo de una función. Aquí tienes un ejemplo:

```
function canYouSpotTheProblem() {  
  "use strict";  
  for (counter = 0; counter < 10; counter++) {  
    console.log("Happy happy");  
  }  
}  
  
canYouSpotTheProblem();  
// → ReferenceError: counter is not defined
```

Normalmente, cuando olvidas poner `let` frente a tu enlace, como en el caso de `counter` en el ejemplo, JavaScript silenciosamente crea un enlace global y lo utiliza. En modo estricto, se reporta un error en su lugar. Esto es muy útil. Sin embargo, cabe mencionar que esto no funciona cuando el enlace en cuestión ya existe en algún lugar del ámbito. En ese caso, el bucle seguirá sobrescribiendo silenciosamente el valor del enlace.

Otro cambio en el modo estricto es que el enlace `this` mantiene el valor `undefined` en funciones que no son llamadas como métodos. Al hacer una llamada de este tipo fuera del modo estricto, `this` se refiere al objeto de ámbito global, que es un objeto cuyas propiedades son los enlaces globales. Entonces, si accidentalmente

llamas incorrectamente a un método o constructor en modo estricto, JavaScript producirá un error tan pronto como intente leer algo de `this`, en lugar de escribir felizmente en el ámbito global.

Por ejemplo, considera el siguiente código, que llama a una función constructor sin la palabra clave `new` para que su `this` *no* se refiera a un objeto recién construido:

```
function Person(name) { this.name = name; }  
let ferdinand = Person("Ferdinand"); // oops  
console.log(name);  
// → Ferdinand
```

Entonces, la llamada falsa a `Person` tuvo éxito pero devolvió un valor no definido y creó el enlace global `name`. En modo estricto, el resultado es diferente.

```
"use strict";  
function Person(name) { this.name = name; }  
let ferdinand = Person("Ferdinand"); // olvidó el new  
// → TypeError: Cannot set property 'name' of undefined
```

Inmediatamente se nos informa que algo está mal. Esto es útil.

Afortunadamente, los constructores creados con la notación `class` siempre mostrarán una queja si se llaman sin `new`, lo que hace que esto sea menos problemático incluso en modo no estricto.

El modo estricto hace algunas cosas más. Prohíbe darle a una función múltiples parámetros con el mismo nombre y elimina ciertas características problemáticas del lenguaje por completo (como la declaración `with`, que es tan incorrecta que no se discute más en este libro).

En resumen, colocar `"use strict"` al principio de tu programa rara vez duele y podría ayudarte a identificar un problema.

TIPOS

Algunos lenguajes quieren saber los tipos de todos tus enlaces y expresiones antes de ejecutar un programa. Te indicarán de inmediato cuando un tipo se utiliza de manera inconsistente. JavaScript considera los tipos solo cuando realmente se ejecuta el programa, e incluso allí a menudo intenta convertir valores implícitamente al tipo que espera, por lo que no es de mucha ayuda.

No obstante, los tipos proporcionan un marco útil para hablar sobre programas. Muchos errores provienen de estar confundido acerca del tipo de valor que entra o sale de una función. Si tienes esa información escrita, es menos probable que te confundas. Podrías agregar un comentario como el siguiente antes de la función `findRoute` del capítulo anterior para describir su tipo:

```
// (graph: Object, from: string, to: string) => string[]
function findRoute(graph, from, to) {
  // ...
}
```

Existen varias convenciones diferentes para anotar programas de JavaScript con tipos.

Una cosa sobre los tipos es que necesitan introducir su propia complejidad para poder describir suficiente código para ser útiles. ¿Qué tipo crees que tendría la función `randomPick` que devuelve un elemento aleatorio de un array? Necesitarías introducir una *variable de tipo*, *T*, que pueda representar cualquier tipo, para que

puedas darle a `randomPick` un tipo como $(T[]) \rightarrow T$ (función de un array de T a un T).

Cuando los tipos de un programa son conocidos, es posible que la computadora los *verifique* por ti, señalando errores antes de que se ejecute el programa. Hay varios dialectos de JavaScript que añaden tipos al lenguaje y los verifican. El más popular se llama [TypeScript](#). Si estás interesado en agregar más rigor a tus programas, te recomiendo que lo pruebes.

En este libro, continuaremos utilizando código JavaScript crudo, peligroso y sin tipos.

PRUEBAS

Si el lenguaje no nos va a ayudar mucho a encontrar errores, tendremos que encontrarlos a la antigua: ejecutando el programa y viendo si hace lo correcto.

Hacer esto manualmente, una y otra vez, es una idea muy mala. No solo es molesto, también tiende a ser ineficaz, ya que lleva demasiado tiempo probar exhaustivamente todo cada vez que haces un cambio.

Las computadoras son buenas en tareas repetitivas, y las pruebas son la tarea repetitiva ideal. Las pruebas automatizadas son el proceso de escribir un programa que prueba otro programa. Es un poco más trabajo escribir pruebas que probar manualmente, pero una vez que lo has hecho, adquieres una especie de superpoder: solo te llevará unos segundos verificar que tu programa siga comportándose correctamente en todas las situaciones para las que escribiste

pruebas. Cuando rompes algo, lo notarás de inmediato en lugar de encontrártelo al azar en algún momento posterior.

Las pruebas suelen tomar la forma de pequeños programas etiquetados que verifican algún aspecto de tu código. Por ejemplo, un conjunto de pruebas para el (probablemente ya probado por alguien más) método `toUpperCase` estándar podría lucir así:

```
function test(label, body) {
  if (!body()) console.log(`Fallo: ${label}`);
}

test("convertir texto latino a mayúsculas", () => {
  return "hello".toUpperCase() == "HELLO";
});
test("convertir texto griego a mayúsculas", () => {
  return "Χαίρετε".toUpperCase() == "XAIPETE";
});
test("no convertir caracteres sin caso", () => {
  return "مرحبا".toUpperCase() == "مرحبا";
});
```

Escribir pruebas de esta forma tiende a producir código bastante repetitivo y torpe. Afortunadamente, existen software que te ayudan a construir y ejecutar colecciones de pruebas (*suites de pruebas*) al proporcionar un lenguaje (en forma de funciones y métodos) adecuado para expresar pruebas y al producir información informativa cuando una prueba falla. Estos suelen llamarse *corredores de pruebas*.

Alguno código es más fácil de probar que otro código. Generalmente, cuantos más objetos externos interactúan con el código, más difícil es configurar el contexto para probarlo. El estilo de programación mostrado en el [capítulo anterior](#), que utiliza valores persistentes

autocontenidos en lugar de objetos cambiantes, tiende a ser fácil de probar.

DEPURACIÓN

Una vez que notas que hay algo mal en tu programa porque se comporta de manera incorrecta o produce errores, el siguiente paso es descubrir *cuál* es el problema.

A veces es obvio. El mensaje de error señalará una línea específica de tu programa, y si miras la descripción del error y esa línea de código, a menudo puedes ver el problema.

Pero no siempre. A veces la línea que desencadenó el problema es simplemente el primer lugar donde se utiliza de manera incorrecta un valor defectuoso producido en otro lugar. Si has estado resolviendo los ejercicios en capítulos anteriores, probablemente ya hayas experimentado estas situaciones.

El siguiente programa de ejemplo intenta convertir un número entero en una cadena en una base dada (decimal, binaria, y así sucesivamente) al seleccionar repetidamente el último dígito y luego dividir el número para deshacerse de este dígito. Pero la extraña salida que produce actualmente sugiere que tiene un error.

```
function numberToString(n, base = 10) {  
  let result = "", sign = "";  
  if (n < 0) {  
    sign = "-";  
    n = -n;  
  }  
  do {  
    result = String(n % base) + result;  
    n /= base;  
  }  
}
```

```

    } while (n > 0);
    return sign + result;
}
console.log(numberToString(13, 10));
// → 1.5e-3231.3e-3221.3e-3211.3e-3201.3e-3191.3e-3181.3...

```

Incluso si ya ves el problema, finge por un momento que no lo haces. Sabemos que nuestro programa no funciona correctamente, y queremos descubrir por qué.

Aquí es donde debes resistir la tentación de empezar a hacer cambios aleatorios en el código para ver si eso lo mejora. En cambio, *piensa*. Analiza lo que está sucediendo y elabora una teoría sobre por qué podría estar ocurriendo. Luego, realiza observaciones adicionales para probar esta teoría, o si aún no tienes una teoría, realiza observaciones adicionales para ayudarte a crear una.

Colocar algunas llamadas `console.log` estratégicas en el programa es una buena manera de obtener información adicional sobre lo que está haciendo el programa. En este caso, queremos que `n` tome los valores 13, 1 y luego 0. Vamos a escribir su valor al inicio del ciclo.

```

13
1.3
0.13
0.013
...
1.5e-323

```

Correcto. Al dividir 13 por 10 no se produce un número entero. En lugar de `n /= base`, lo que realmente queremos es `n = Math.floor(n / base)` para que el número se “desplace” correctamente hacia la derecha.

Una alternativa a usar `console.log` para observar el comportamiento del programa es utilizar las capacidades del *depurador* de tu navegador. Los navegadores vienen con la capacidad de establecer un *punto de interrupción* en una línea específica de tu código. Cuando la ejecución del programa llega a una línea con un punto de interrupción, se pausa y puedes inspeccionar los valores de las asignaciones en ese punto. No entraré en detalles, ya que los depuradores difieren de un navegador a otro, pero busca en las herramientas de desarrollo de tu navegador o busca instrucciones en la Web. Otra forma de establecer un punto de interrupción es incluir una instrucción `debugger` (consistente únicamente en esa palabra clave) en tu programa. Si las herramientas de desarrollo de tu navegador están activas, el programa se pausará cada vez que alcance dicha instrucción.

PROPAGACIÓN DE ERRORES

Lamentablemente, no todos los problemas pueden ser prevenidos por el programador. Si tu programa se comunica de alguna manera con el mundo exterior, es posible recibir entradas malformadas, sobrecargarse de trabajo o que falle la red.

Si estás programando solo para ti, puedes permitirte simplemente ignorar esos problemas hasta que ocurran. Pero si estás construyendo algo que será utilizado por alguien más, generalmente quieres que el programa haga algo más que simplemente colapsar. A veces lo correcto es aceptar la entrada incorrecta y continuar ejecutándose. En otros casos, es mejor informar al usuario sobre lo que salió mal y luego rendirse. Pero en cualquier situación, el programa debe hacer algo activamente en respuesta al problema.

Imaginemos que tienes una función `promptNumber` que solicita al usuario un número y lo retorna. ¿Qué debería retornar si el usuario ingresa “naranja”?

Una opción es hacer que retorne un valor especial. Las opciones comunes para tales valores son `null`, `undefined` o `-1`.

```
function promptNumber(pregunta) {  
  let resultado = Number(prompt(pregunta));  
  if (Number.isNaN(resultado)) return null;  
  else return resultado;  
}  
  
console.log(promptNumber("¿Cuántos árboles ves?"));
```

Ahora, cualquier código que llame a `promptNumber` debe verificar si se leyó un número real y, de no ser así, debe recuperarse de alguna manera, quizás volviendo a preguntar o completando con un valor predeterminado. O podría retornar nuevamente un valor especial a su llamante para indicar que no pudo hacer lo que se le pidió.

En muchas situaciones, sobre todo cuando los errores son comunes y el llamante debería tomarlos explícitamente en cuenta, retornar un valor especial es una buena manera de indicar un error. Sin embargo, tiene sus inconvenientes. Primero, ¿qué pasa si la función ya puede devolver todos los tipos posibles de valores? En tal función, tendrás que hacer algo como envolver el resultado en un objeto para poder distinguir el éxito del fracaso, de la misma manera que lo hace el método `next` en la interfaz del iterador.

```
function lastElement(arreglo) {  
  if (arreglo.length == 0) {  
    return {falló: true};  
  } else {
```

```
    return {valor: arreglo[arreglo.length - 1]};  
  }  
}
```

El segundo problema con retornar valores especiales es que puede llevar a un código incómodo. Si un fragmento de código llama a `promptNumber` 10 veces, tendrá que verificar 10 veces si se devolvió `null`. Y si su respuesta al encontrar `null` es simplemente devolver `null` en sí mismo, los llamantes de la función a su vez tendrán que comprobarlo, y así sucesivamente.

EXCEPCIONES

Cuando una función no puede proceder normalmente, lo que a menudo *queremos* hacer es simplemente detener lo que estamos haciendo e ir directamente a un lugar que sepa cómo manejar el problema. Esto es lo que hace el *manejo de excepciones*.

Las excepciones son un mecanismo que hace posible que el código que se encuentra con un problema *lanze* (o *emita*) una excepción. Una excepción puede ser cualquier valor. Lanzar una se asemeja de alguna manera a un retorno super potenciado de una función: sale no solo de la función actual sino también de sus llamadores, hasta llegar a la primera llamada que inició la ejecución actual. Esto se llama *desenrollar la pila*. Puede recordar la pila de llamadas a funciones que se mencionó en el [Capítulo 3](#). Una excepción recorre esta pila, descartando todos los contextos de llamada que encuentra.

Si las excepciones siempre fueran directamente hasta el final de la pila, no serían de mucha utilidad. Simplemente proporcionarían una forma novedosa de hacer que su programa falle. Su poder radica en

el hecho de que puede colocar “obstáculos” a lo largo de la pila para *capturar* la excepción mientras viaja hacia abajo. Una vez que ha capturado una excepción, puede hacer algo con ella para resolver el problema y luego continuar ejecutando el programa.

Aquí tienes un ejemplo:

```
function promptDirection(question) {
  let result = prompt(question);
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new Error("Dirección inválida: " + result);
}

function look() {
  if (promptDirection("¿Hacia dónde?") == "L") {
    return "una casa";
  } else {
    return "dos osos enojados";
  }
}

try {
  console.log("Ves", look());
} catch (error) {
  console.log("Algo salió mal: " + error);
}
```

La palabra clave `throw` se utiliza para lanzar una excepción. La captura de una excepción se realiza envolviendo un trozo de código en un bloque `try`, seguido de la palabra clave `catch`. Cuando el código en el bloque `try` provoca que se lance una excepción, se evalúa el bloque `catch`, con el nombre entre paréntesis vinculado al valor de la excepción. Después de que el bloque `catch` finalice, o si el bloque `try` finaliza sin problemas, el programa continúa debajo de toda la instrucción `try/catch`.

En este caso, utilizamos el constructor `Error` para crear nuestro valor de excepción. Este es un constructor de JavaScript estándar que crea un objeto con una propiedad `message`. Las instancias de `Error` también recopilan información sobre la pila de llamadas que existía cuando se creó la excepción, una llamada *traza de pila*. Esta información se almacena en la propiedad `stack` y puede ser útil al intentar depurar un problema: nos indica la función donde ocurrió el problema y qué funciones realizaron la llamada fallida.

Ten en cuenta que la función `look` ignora por completo la posibilidad de que `promptDirection` pueda fallar. Esta es la gran ventaja de las excepciones: el código de manejo de errores solo es necesario en el punto donde ocurre el error y en el punto donde se maneja. Las funciones intermedias pueden olvidarse por completo de ello.

Bueno, casi...

LIMPIANDO DESPUÉS DE EXCEPCIONES

El efecto de una excepción es otro tipo de flujo de control. Cada acción que pueda causar una excepción, que es prácticamente cada llamada a función y acceso a propiedad, puede hacer que el control salga repentinamente de tu código.

Esto significa que cuando el código tiene varios efectos secundarios, incluso si su flujo de control “regular” parece que siempre ocurrirán todos, una excepción podría evitar que algunos de ellos sucedan.

Aquí tienes un código bancario realmente malo.

```

const accounts = {
  a: 100,
  b: 0,
  c: 20
};

function getAccount() {
  let accountName = prompt("Ingresa el nombre de una cuenta");
  if (!Object.hasOwn(accounts, accountName)) {
    throw new Error(`No existe esa cuenta: ${accountName}`);
  }
  return accountName;
}

function transfer(from, amount) {
  if (accounts[from] < amount) return;
  accounts[from] -= amount;
  accounts[getAccount()] += amount;
}

```

La función `transfer` transfiere una suma de dinero desde una cuenta dada a otra, pidiendo el nombre de la otra cuenta en el proceso. Si se proporciona un nombre de cuenta inválido, `getAccount` lanza una excepción.

Pero `transfer` *primero* retira el dinero de la cuenta y *luego* llama a `getAccount` antes de agregarlo a otra cuenta. Si se interrumpe por una excepción en ese momento, simplemente hará desaparecer el dinero.

Ese código podría haber sido escrito de manera un poco más inteligente, por ejemplo, llamando a `getAccount` antes de comenzar a mover el dinero. Pero a menudo los problemas como este ocurren de formas más sutiles. Incluso las funciones que no parecen que lanzarán una excepción podrían hacerlo en circunstancias excepcionales o cuando contienen un error del programador.

Una manera de abordar esto es utilizar menos efectos secundarios. Nuevamente, un estilo de programación que calcule nuevos valores en lugar de cambiar datos existentes ayuda. Si un fragmento de código deja de ejecutarse en medio de la creación de un nuevo valor, no se dañaron estructuras de datos existentes, lo que facilita la recuperación.

Pero eso no siempre es práctico. Por eso existe otra característica que tienen las instrucciones `try`. Pueden estar seguidas de un bloque `finally` en lugar o además de un bloque `catch`. Un bloque `finally` dice “sin importar *qué* suceda, ejecuta este código después de intentar ejecutar el código en el bloque `try`.”

```
function transfer(from, amount) {
  if (accounts[from] < amount) return;
  let progress = 0;
  try {
    accounts[from] -= amount;
    progress = 1;
    accounts[getAccount()] += amount;
    progress = 2;
  } finally {
    if (progress == 1) {
      accounts[from] += amount;
    }
  }
}
```

Esta versión de la función rastrea su progreso y, si al salir nota que fue abortada en un punto donde había creado un estado del programa inconsistente, repara el daño causado.

Cabe destacar que aunque el código `finally` se ejecuta cuando se lanza una excepción en el bloque `try`, no interfiere con la excepción.

Después de que se ejecuta el bloque `finally`, la pila continúa desenrollándose.

Escribir programas que funcionen de manera confiable incluso cuando surgen excepciones en lugares inesperados es difícil. Muchas personas simplemente no se preocupan, y debido a que las excepciones suelen reservarse para circunstancias excepcionales, el problema puede ocurrir tan raramente que ni siquiera se note. Si eso es algo bueno o realmente malo depende de cuánto daño causará el software cuando falle.

CAPTURA SELECTIVA

Cuando una excepción llega hasta el final de la pila sin ser capturada, es manejada por el entorno. Lo que esto significa difiere según los entornos. En los navegadores, generalmente se escribe una descripción del error en la consola de JavaScript (accesible a través del menú Herramientas o Desarrollador del navegador). Node.js, el entorno de JavaScript sin navegador del que hablaremos en [Capítulo 20](#), es más cuidadoso con la corrupción de datos. Abortará todo el proceso cuando ocurra una excepción no manejada.

Para errores de programación, a menudo dejar que el error siga su curso es lo mejor que se puede hacer. Una excepción no manejada es una forma razonable de señalar un programa defectuoso, y la consola de JavaScript proporcionará, en navegadores modernos, información sobre qué llamadas a funciones estaban en la pila cuando ocurrió el problema.

Para problemas que se *espera* que ocurran durante el uso rutinario, fallar con una excepción no manejada es una estrategia terrible.

Usos incorrectos del lenguaje, como hacer referencia a un enlace inexistente, buscar una propiedad en `null` o llamar a algo que no es una función, también provocarán que se lancen excepciones. Estas excepciones también pueden ser capturadas.

Cuando se entra en un cuerpo `catch`, todo lo que sabemos es que *algo* en nuestro cuerpo `try` causó una excepción. Pero no sabemos *qué* lo hizo ni *qué* excepción causó.

JavaScript (en una omisión bastante llamativa) no proporciona un soporte directo para capturar excepciones selectivamente: o las capturas todas o no capturas ninguna. Esto hace que sea tentador *asumir* que la excepción que obtienes es la que tenías en mente cuando escribiste el bloque `catch`.

Pero podría no serlo. Alguno otra asunción podría estar violada, o podrías haber introducido un error que está causando una excepción. Aquí tienes un ejemplo que *intenta* seguir llamando a `promptDirection` hasta obtener una respuesta válida:

```
for (;;) {
  try {
    let dir = promptDirection("¿Dónde?"); // ← ¡Error de tipeo!
    console.log("Elegiste ", dir);
    break;
  } catch (e) {
    console.log("Dirección no válida. Inténtalo de nuevo.");
  }
}
```

La construcción `for (;;)` es una forma de crear intencionalmente un bucle que no se termina por sí mismo. Salimos del bucle solo cuando se proporciona una dirección válida. *Pero* escribimos mal

`promptDirection`, lo que resultará en un error de “variable no definida”. Debido a que el bloque `catch` ignora por completo el valor de la excepción (`e`), asumiendo que sabe cuál es el problema, trata erróneamente el error de enlace mal escrito como indicativo de una entrada incorrecta. Esto no solo causa un bucle infinito, sino que también “entorpece” el útil mensaje de error sobre el enlace mal escrito.

Como regla general, no captures excepciones de manera general a menos que sea con el propósito de “enviarlas” a algún lugar, por ejemplo, a través de la red para informar a otro sistema que nuestro programa se bloqueó. E incluso en ese caso, piensa cuidadosamente cómo podrías estar ocultando información.

Por lo tanto, queremos capturar un tipo *específico* de excepción. Podemos hacer esto verificando en el bloque `catch` si la excepción que recibimos es la que nos interesa y relanzándola en caso contrario. Pero, ¿cómo reconocemos una excepción?

Podríamos comparar su propiedad `message` con el mensaje que esperamos error. Pero esta es una forma poco confiable de escribir código, estaríamos utilizando información diseñada para consumo humano (el mensaje) para tomar una decisión programática. Tan pronto como alguien cambie (o traduzca) el mensaje, el código dejará de funcionar.

En lugar de eso, definamos un nuevo tipo de error y usemos `instanceof` para identificarlo.

```
class InputError extends Error {}  
  
function promptDirection(question) {
```

```

    let result = prompt(question);
    if (result.toLowerCase() == "izquierda") return "I";
    if (result.toLowerCase() == "derecha") return "D";
    throw new InputError("Dirección no válida: " + result);
}

```

La nueva clase de error extiende `Error`. No define su propio constructor, lo que significa que hereda el constructor de `Error`, que espera un mensaje de cadena como argumento. De hecho, no define nada en absoluto, la clase está vacía. Los objetos `InputError` se comportan como objetos `Error`, excepto que tienen una clase diferente mediante la cual podemos reconocerlos.

Ahora el bucle puede capturar esto con más cuidado.

```

for (;;) {
  try {
    let dir = promptDirection("¿Dónde?");
    console.log("Elegiste ", dir);
    break;
  } catch (e) {
    if (e instanceof InputError) {
      console.log("Dirección no válida. Inténtalo de nuevo.");
    } else {
      throw e;
    }
  }
}
}

```

Esto capturará solo instancias de `InputError` y permitirá que pasen excepciones no relacionadas. Si vuelves a introducir el error de tipo, el error de enlace no definido se informará correctamente.

AFIRMACIONES

Las *afirmaciones* son verificaciones dentro de un programa que aseguran que algo es como se supone que debe ser. Se utilizan no para manejar situaciones que pueden surgir en la operación normal, sino para encontrar errores de programación.

Si, por ejemplo, se describe `primerElemento` como una función que nunca debería ser llamada en arrays vacíos, podríamos escribirla de la siguiente manera:

```
function primerElemento(array) {  
  if (array.length == 0) {  
    throw new Error("primerElemento llamado con []");  
  }  
  return array[0];  
}
```

Ahora, en lugar de devolver silenciosamente `undefined` (que es lo que obtienes al leer una propiedad de un array que no existe), esto hará que tu programa falle ruidosamente tan pronto como lo uses incorrectamente. Esto hace que sea menos probable que tales errores pasen desapercibidos y más fácil encontrar su causa cuando ocurran.

No recomiendo intentar escribir afirmaciones para cada tipo de entrada incorrecta posible. Eso sería mucho trabajo y llevaría a un código muy ruidoso. Querrás reservarlas para errores que son fáciles de cometer (o que te encuentres cometiendo).

RESUMEN

Una parte importante de programar es encontrar, diagnosticar y corregir errores. Los problemas pueden ser más fáciles de notar si

tienes un conjunto de pruebas automatizadas o agregas afirmaciones a tus programas.

Los problemas causados por factores fuera del control del programa generalmente deberían ser planificados activamente. A veces, cuando el problema puede ser manejado localmente, los valores de retorno especiales son una buena forma de rastrearlos. De lo contrario, las excepciones pueden ser preferibles.

Lanzar una excepción provoca que la pila de llamadas se desenrolle hasta el próximo bloque `try/catch` envolvente o hasta la base de la pila. El valor de la excepción será entregado al bloque `catch` que la captura, el cual debe verificar que sea realmente el tipo de excepción esperado y luego hacer algo con él. Para ayudar a abordar el flujo de control impredecible causado por las excepciones, se pueden utilizar bloques `finally` para asegurar que un trozo de código se ejecute *siempre* cuando un bloque termina.

EJERCICIOS

REINTENTAR

Imagina que tienes una función `primitiveMultiply` que en el 20 por ciento de los casos multiplica dos números y en el otro 80 por ciento arroja una excepción del tipo

`MultiplierUnitFailure`. Escribe una función que envuelva esta función problemática y siga intentando hasta que una llamada tenga éxito, momento en el que devuelva el resultado.

Asegúrate de manejar solo las excepciones que estás intentando manejar.

LA CAJA CERRADA CON LLAVE

Considera el siguiente objeto (bastante artificial):

```
const box = new class {  
  locked = true;  
  #content = [];  
  
  unlock() { this.locked = false; }  
  lock() { this.locked = true; }  
  get content() {  
    if (this.locked) throw new Error("¡Cerrado con llave!");  
    return this.#content;  
  }  
};
```

Es una caja con una cerradura. Hay un array en la caja, pero solo puedes acceder a él cuando la caja está desbloqueada.

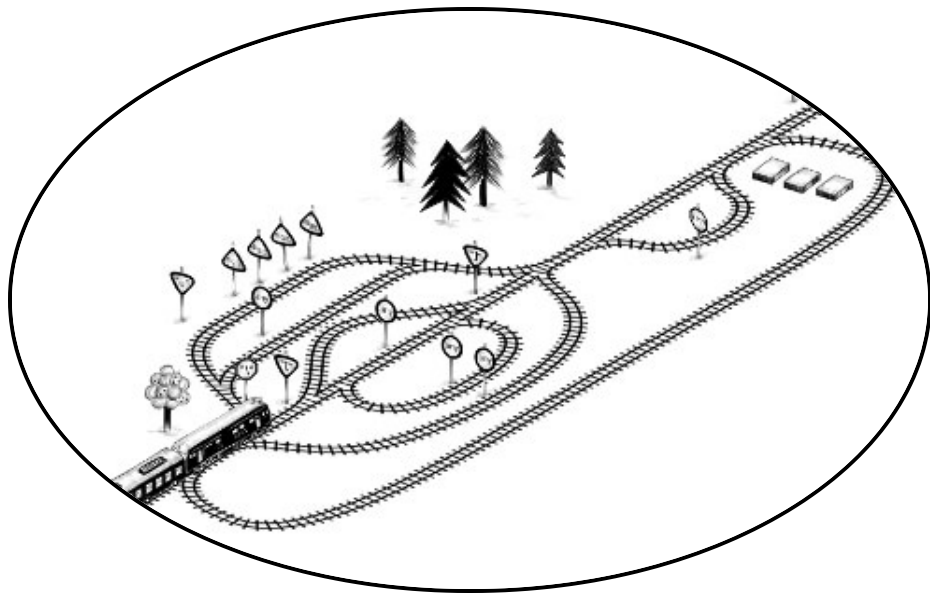
Escribe una función llamada `withBoxUnlocked` que reciba como argumento un valor de función, desbloquee la caja, ejecute la función y luego asegure que la caja esté cerrada de nuevo antes de devolverla, independientemente de si la función de argumento devolvió normalmente o lanzó una excepción.

Para puntos adicionales, asegúrate de que si llamas a `withBoxUnlocked` cuando la caja ya está desbloqueada, la caja permanezca desbloqueada.

EXPRESIONES REGULARES

“Algunas personas, cuando se enfrentan a un problema, piensan '¡Ya sé, usaré expresiones regulares!' Ahora tienen dos problemas.”

—Jamie Zawinski



Las herramientas y técnicas de programación sobreviven y se propagan de manera caótica y evolutiva. No siempre ganan las mejores o brillantes, sino aquellas que funcionan lo suficientemente bien dentro del nicho correcto o que se integran con otra pieza exitosa de tecnología.

En este capítulo, discutiré una de esas herramientas, *expresiones regulares*. Las expresiones regulares son una forma de describir

patrones en datos de cadena. Forman un pequeño lenguaje separado que es parte de JavaScript y muchos otros lenguajes y sistemas.

Las expresiones regulares son tanto terriblemente incómodas como extremadamente útiles. Su sintaxis es críptica y la interfaz de programación que JavaScript proporciona para ellas es torpe. Pero son una herramienta poderosa para inspeccionar y procesar cadenas. Comprender adecuadamente las expresiones regulares te hará un programador más efectivo.

CREANDO UNA EXPRESIÓN REGULAR

Una expresión regular es un tipo de objeto. Puede ser construido con el constructor `RegExp` o escrito como un valor literal al encerrar un patrón entre caracteres de barra diagonal (`/`).

```
let re1 = new RegExp("abc");  
let re2 = /abc/;
```

Ambos objetos de expresión regular representan el mismo patrón: un carácter *a* seguido de un *b* seguido de un *c*.

Cuando se utiliza el constructor `RegExp`, el patrón se escribe como una cadena normal, por lo que se aplican las reglas habituales para las barras invertidas.

La segunda notación, donde el patrón aparece entre caracteres de barra diagonal, trata las barras invertidas de manera un poco diferente. Primero, dado que una barra diagonal termina el patrón, debemos poner una barra invertida antes de cualquier barra diagonal que queramos que sea *parte* del patrón. Además, las barras invertidas que no forman parte de códigos de caracteres especiales

(como `\n`) serán *preservadas*, en lugar de ser ignoradas como lo son en las cadenas, y cambian el significado del patrón. Algunos caracteres, como signos de interrogación y signos de más, tienen significados especiales en las expresiones regulares y deben ser precedidos por una barra invertida si se desea representar el propio carácter.

```
let aPlus = /A\+/;
```

PRUEBAS DE COINCIDENCIAS

Los objetos de expresiones regulares tienen varios métodos. El más simple es `test`. Si le pasas una cadena, devolverá un Booleano indicándote si la cadena contiene una coincidencia con el patrón de la expresión.

```
console.log(/abc/.test("abcde"));  
// → true  
console.log(/abc/.test("abxde"));  
// → false
```

Una expresión regular que consiste solo en caracteres no especiales simplemente representa esa secuencia de caracteres. Si *abc* aparece en cualquier parte de la cadena contra la cual estamos probando (no solo al principio), `test` devolverá `true`.

CONJUNTOS DE CARACTERES

Descubrir si una cadena contiene *abc* también se podría hacer con una llamada a `indexOf`. Las expresiones regulares son útiles porque nos permiten describir patrones más complicados.

Digamos que queremos hacer coincidir cualquier número. En una expresión regular, poner un conjunto de caracteres entre corchetes hace que esa parte de la expresión coincida con cualquiera de los caracteres entre los corchetes.

Ambas expresiones siguientes hacen coincidir todas las cadenas que contienen un dígito:

```
console.log(/[0123456789]/.test("in 1992"));  
// → true  
console.log(/[0-9]/.test("in 1992"));  
// → true
```

Dentro de corchetes, un guion (-) entre dos caracteres se puede usar para indicar un rango de caracteres, donde el orden es determinado por el número del carácter en el Unicode. Los caracteres del 0 al 9 están uno al lado del otro en este orden (códigos 48 a 57), por lo que `[0-9]` abarca todos ellos y coincide con cualquier dígito.

Varios grupos comunes de caracteres tienen sus propias abreviaturas incorporadas. Los dígitos son uno de ellos: `\d` significa lo mismo que `[0-9]`.

- `\d` Cualquier carácter dígito
- `\w` Un carácter alfanumérico (“carácter de palabra”)
- `\s` Cualquier carácter de espacio en blanco (espacio, tabulación, nueva línea, y similares)
- `\D` Un carácter que *no* es un dígito
- `\W` Un carácter no alfanumérico
- `\S` Un carácter que no es de espacio en blanco
- `.` Cualquier carácter excepto nueva línea

Así que podrías hacer coincidir un formato de fecha y hora como 01-30-2003 15:20 con la siguiente expresión:

```
let dateTime = /\d\d-\d\d-\d\d\d\d \d\d:\d\d/;
console.log(dateTime.test("01-30-2003 15:20"));
// → true
console.log(dateTime.test("30-ene-2003 15:20"));
// → false
```

¡Eso se ve completamente horrible, ¿verdad? La mitad son barras invertidas, produciendo un ruido de fondo que dificulta identificar el patrón expresado. Veremos una versión ligeramente mejorada de esta expresión [más adelante](#).

Estos códigos de barra invertida también se pueden usar dentro de corchetes. Por ejemplo, `[\d.]` significa cualquier dígito o un carácter de punto. Pero el punto en sí, entre corchetes, pierde su significado especial. Lo mismo ocurre con otros caracteres especiales, como `+`.

Para *invertir* un conjunto de caracteres, es decir, expresar que deseas hacer coincidir cualquier carácter *excepto* los que están en el conjunto, puedes escribir un carácter circunflejo (^) después del corchete de apertura.

```
let nonBinary = /^[^01]/;
console.log(nonBinary.test("1100100010100110"));
// → false
console.log(nonBinary.test("0111010112101001"));
// → true
```

CARACTERES INTERNACIONALES

Debido a la implementación simplista inicial de JavaScript y al hecho de que este enfoque simplista luego se estableció como comportamiento estándar, las expresiones regulares de JavaScript son bastante simples en lo que respecta a los caracteres que no aparecen en el idioma inglés. Por ejemplo, según las expresiones regulares de JavaScript, un “carácter de palabra” es solo uno de los 26 caracteres del alfabeto latino (mayúsculas o minúsculas), dígitos decimales y, por alguna razón, el guion bajo. Cosas como *é* o *β*, que definitivamente son caracteres de palabra, no coincidirán con `\w` (y sí coincidirán con `\W` en mayúsculas, la categoría de no palabras).

Por un extraño accidente histórico, `\s` (espacio en blanco) no tiene este problema y coincide con todos los caracteres que el estándar Unicode considera espacios en blanco, incluidos elementos como el espacio sin ruptura y el separador de vocal mongol.

Es posible usar `\p` en una expresión regular para hacer coincidir todos los caracteres a los que el estándar Unicode asigna una propiedad dada. Esto nos permite hacer coincidir cosas como letras de una manera más cosmopolita. Sin embargo, nuevamente debido a la compatibilidad con los estándares originales del lenguaje, estos solo se reconocen cuando se coloca un carácter `u` (por Unicode) después de la expresión regular.

<code>\p{L}</code>	Cualquier letra
<code>\p{N}</code>	Cualquier carácter numérico
<code>\p{P}</code>	Cualquier carácter de puntuación
<code>\P{L}</code>	Cualquier no letra (la P en mayúsculas invierte)
<code>\p{Script=Hangul}</code>	Cualquier carácter del guion dado (ver

Capítulo 5)

Usar `\w` para el procesamiento de texto que puede necesitar manejar texto no inglés (o incluso texto en inglés con palabras prestadas como “cliché”) es una desventaja, ya que no tratará caracteres como “é” como letras. Aunque tienden a ser un poco más verbosos, los grupos de propiedades `\p` son más robustos.

```
console.log(/\p{L}/u.test("α"));
// → true
console.log(/\p{L}/u.test("!"));
// → false
console.log(/\p{Script=Greek}/u.test("α"));
// → true
console.log(/\p{Script=Arabic}/u.test("α"));
// → false
```

Por otro lado, si estás haciendo coincidir números para hacer algo con ellos, a menudo querrás usar `\d` para dígitos, ya que convertir caracteres numéricos arbitrarios en un número de JavaScript no es algo que una función como `Number` pueda hacer por ti.

REPETIR PARTES DE UN PATRÓN

Ahora sabemos cómo hacer coincidir un solo dígito. ¿Qué tal si queremos hacer coincidir un número entero, una secuencia de uno o más dígitos?

Cuando colocas un signo más (+) después de algo en una expresión regular, indica que el elemento puede repetirse más de una vez. Así, `/\d+/` hace coincidir uno o más caracteres de dígitos.

```
console.log(/\d+/.test("'123'"));
// → true
```

```
console.log(/\d+/.test(''));  
// → false  
console.log(/\d*/.test('123'));  
// → true  
console.log(/\d*/.test(''));  
// → true
```

El asterisco (*) tiene un significado similar pero también permite que el patrón coincida cero veces. Algo con un asterisco después nunca impide que un patrón coincida, simplemente coincidirá cero veces si no puede encontrar ningún texto adecuado para hacer coincidir.

Un signo de interrogación hace que una parte de un patrón sea *opcional*, lo que significa que puede ocurrir cero veces o una vez. En el siguiente ejemplo, se permite que el carácter *u* ocurra, pero el patrón también coincide cuando falta.

```
let neighbor = /neighbou?r/;  
console.log(neighbor.test("neighbour"));  
// → true  
console.log(neighbor.test("neighbor"));  
// → true
```

Para indicar que un patrón debe ocurrir un número preciso de veces, utiliza llaves. Colocar {4} después de un elemento, por ejemplo, requiere que ocurra exactamente cuatro veces. También es posible especificar un rango de esta manera: {2, 4} significa que el elemento debe ocurrir al menos dos veces y como máximo cuatro veces.

Aquí tienes otra versión del patrón de fecha y hora que permite días, meses y horas de uno o dos dígitos. También es un poco más fácil de entender.


```
let dateTime = /\d{1,2}-\d{1,2}-\d{4} \d{1,2}:\d{2}/;  
console.log(dateTime.test("1-30-2003 8:45"));  
// → true
```

También puedes especificar rangos abiertos al utilizar llaves omitiendo el número después de la coma. Así, `{5,}` significa cinco o más veces.

AGRUPACIÓN DE SUBEXPRESIONES

Para usar un operador como `*` o `+` en más de un elemento a la vez, debes utilizar paréntesis. Una parte de una expresión regular que está encerrada entre paréntesis cuenta como un solo elemento en lo que respecta a los operadores que le siguen.

```
let cartoonCrying = /boo+(hoo+)+/i;  
console.log(cartoonCrying.test("Boohooooohooohoo"));  
// → true
```

Los primeros y segundos caracteres `+` aplican solo al segundo `o` en *boo* y *hoo*, respectivamente. El tercer `+` se aplica a todo el grupo `(hoo+)`, haciendo coincidir una o más secuencias como esa.

La `i` al final de la expresión en el ejemplo hace que esta expresión regular ignore mayúsculas y minúsculas, lo que le permite hacer coincidir la *B* mayúscula en la cadena de entrada, aunque el patrón en sí está completamente en minúsculas.

COINCIDENCIAS Y GRUPOS

El método `test` es la forma más simple de hacer coincidir una expresión regular. Solo te indica si hubo coincidencia y nada más. Las expresiones regulares también tienen un método `exec`

(ejecutar) que devolverá `null` si no se encontró ninguna coincidencia y devolverá un objeto con información sobre la coincidencia en caso contrario.

```
let coincidencia = /\d+/.exec("uno dos 100");
console.log(coincidencia);
// → ["100"]
console.log(coincidencia.index);
// → 8
```

Un objeto devuelto por `exec` tiene una propiedad de `index` que nos dice *dónde* en la cadena comienza la coincidencia exitosa. Aparte de eso, el objeto parece (y de hecho es) un array de strings, cuyo primer elemento es la cadena que coincidió. En el ejemplo anterior, esta es la secuencia de dígitos que estábamos buscando.

Los valores de tipo string tienen un método `match` que se comporta de manera similar.

```
console.log("uno dos 100".match(/\d+/));
// → ["100"]
```

Cuando la expresión regular contiene subexpresiones agrupadas con paréntesis, el texto que coincidió con esos grupos también aparecerá en el array. La coincidencia completa es siempre el primer elemento. El siguiente elemento es la parte coincidente con el primer grupo (el que tiene el paréntesis de apertura primero en la expresión), luego el segundo grupo, y así sucesivamente.

```
let textoEntreComillas = /^('[^']*')$/;
console.log(textoEntreComillas.exec("ella dijo 'hola'"));
// → ["'hola'", "hola"]
```

Cuando un grupo no termina coincidiendo en absoluto (por ejemplo, cuando está seguido por un signo de pregunta), su posición en el array de salida contendrá `undefined`. Y cuando un grupo coincide múltiples veces (por ejemplo, cuando está seguido por un `+`), solo la última coincidencia termina en el array.

```
console.log(/mal(mente)?/.exec("mal"));  
// → ["mal", undefined]  
console.log(/(\d)+/.exec("123"));  
// → ["123", "3"]
```

Si quieres utilizar paréntesis puramente para agrupar, sin que aparezcan en el array de coincidencias, puedes colocar `?:` después del paréntesis de apertura.

```
console.log(/(?:na)+/.exec("banana"));  
// → ["nana"]
```

Los grupos pueden ser útiles para extraer partes de una cadena. Si no solo queremos verificar si una cadena contiene una fecha sino también extraerla y construir un objeto que la represente, podemos envolver paréntesis alrededor de los patrones de dígitos y seleccionar directamente la fecha del resultado de `exec`.

Pero primero haremos un breve desvío, en el que discutiremos la forma incorporada de representar fechas y horas en JavaScript.

LA CLASE DATE

JavaScript tiene una clase estándar para representar fechas—o, más bien, puntos en tiempo. Se llama `Date`. Si simplemente creas un objeto de fecha usando `new`, obtendrás la fecha y hora actuales.

```
console.log(new Date());  
// → Fri Feb 02 2024 18:03:06 GMT+0100 (CET)
```

También puedes crear un objeto para un momento específico.

```
console.log(new Date(2009, 11, 9));  
// → Mié Dec 09 2009 00:00:00 GMT+0100 (CET)  
console.log(new Date(2009, 11, 9, 12, 59, 59, 999));  
// → Mié Dec 09 2009 12:59:59 GMT+0100 (CET)
```

JavaScript utiliza una convención donde los números de mes empiezan en cero (por lo que diciembre es 11), pero los números de día comienzan en uno. Esto es confuso y tonto. Ten cuidado.

Los últimos cuatro argumentos (horas, minutos, segundos y milisegundos) son opcionales y se consideran cero cuando no se proporcionan.

Las marcas de tiempo se almacenan como el número de milisegundos desde el comienzo de 1970, en UTC (zona horaria). Esto sigue una convención establecida por “tiempo de Unix”, que fue inventado alrededor de esa época. Puedes usar números negativos para tiempos antes de 1970. El método `getTime` en un objeto de fecha retorna este número. Es grande, como te puedes imaginar.

```
console.log(new Date(2013, 11, 19).getTime());  
// → 1387407600000  
console.log(new Date(1387407600000));  
// → Jue Dec 19 2013 00:00:00 GMT+0100 (CET)
```

Si le proporcionas un único argumento al constructor `Date`, ese argumento se tratará como un recuento de milisegundos. Puedes obtener el recuento actual de milisegundos creando un nuevo objeto

Date y llamando a `getTime` en él o llamando a la función `Date.now`.

Los objetos de fecha proporcionan métodos como `getFullYear`, `getMonth`, `getDate`, `getHours`, `getMinutes` y `getSeconds` para extraer sus componentes. Además de `getFullYear`, también existe `getYear`, que te da el año menos 1900 (98 o 119) y es en su mayoría inútil.

Poniendo paréntesis alrededor de las partes de la expresión que nos interesan, podemos crear un objeto de fecha a partir de una cadena.

```
function getDate(string) {  
  let [_ , month, day, year] =  
    /(\d{1,2})-(\d{1,2})-(\d{4})/.exec(string);  
  return new Date(year, month - 1, day);  
}  
console.log(getDate("1-30-2003"));  
// → Tue Jan 30 2003 00:00:00 GMT+0100 (CET)
```

La vinculación `_` (guion bajo) se ignora y se utiliza solo para omitir el elemento de coincidencia completa en el array devuelto por `exec`.

LÍMITES Y ANTICIPACIÓN

Desafortunadamente, `getDate` también extraerá felizmente una fecha de la cadena `"100-1-30000"`. Una coincidencia puede ocurrir en cualquier parte de la cadena, por lo que en este caso, simplemente empezará en el segundo carácter y terminará en el antepenúltimo carácter.

Si queremos asegurar que la coincidencia abarque toda la cadena, podemos agregar los marcadores `^` y `$`. El circunflejo coincide con el

inicio de la cadena de entrada, mientras que el signo de dólar coincide con el final. Por lo tanto, `/^\d+$/` coincide con una cadena que consiste completamente de uno o más dígitos, `/^!/` coincide con cualquier cadena que comience con un signo de exclamación y `/x^/` no coincide con ninguna cadena (no puede haber una `x` antes del inicio de la cadena).

También existe un marcador `\b`, que coincide con los “límites de palabra”, posiciones que tienen un carácter de palabra a un lado y un carácter que no es de palabra al otro. Desafortunadamente, estos utilizan el mismo concepto simplista de caracteres de palabra que `\w`, por lo que no son muy confiables.

Ten en cuenta que estos marcadores no coinciden con ningún carácter real. Simplemente aseguran que se cumpla una condición determinada en el lugar donde aparecen en el patrón.

Las pruebas de *mirar adelante* hacen algo similar. Proporcionan un patrón y harán que la coincidencia falle si la entrada no coincide con ese patrón, pero en realidad no mueven la posición de la coincidencia hacia adelante. Se escriben entre `(?=` y `)`.

```
console.log(/a(?=e)/.exec("braeburn"));  
// → ["a"]  
console.log(/a(?! )/.exec("a b"));  
// → null
```

Observa cómo la `e` en el primer ejemplo es necesaria para coincidir, pero no forma parte de la cadena coincidente. La notación `(?!)` expresa un mirar adelante *negativo*. Esto solo coincide si el patrón entre paréntesis *no* coincide, lo que hace que el segundo ejemplo solo

coincida con caracteres “a” que no tienen un espacio después de ellos.

PATRONES DE ELECCIÓN

Digamos que queremos saber si un texto contiene no solo un número, sino un número seguido de una de las palabras *pig*, *cow* o *chicken*, o cualquiera de sus formas en plural.

Podríamos escribir tres expresiones regulares y probarlas sucesivamente, pero hay una forma más sencilla. El carácter de barra vertical (|) denota una elección entre el patrón a su izquierda y el patrón a su derecha. Así que puedo decir esto:

```
let animalCount = /\d+ (pig|cow|chicken)s?/;  
console.log(animalCount.test("15 pigs"));  
// → true  
console.log(animalCount.test("15 pugs"));  
// → false
```

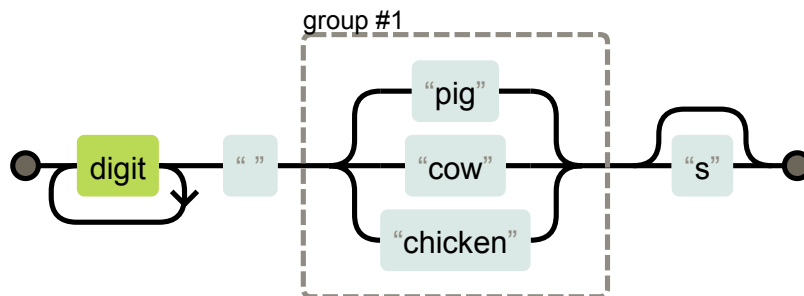
Los paréntesis se pueden utilizar para limitar la parte del patrón a la que se aplica el operador de barra, y puedes colocar varios de estos operadores uno al lado del otro para expresar una elección entre más de dos alternativas.

LA MECÁNICA DE LA COINCIDENCIA

Conceptualmente, cuando utilizas `exec` o `test`, el motor de expresiones regulares busca una coincidencia en tu cadena tratando de ajustar primero la expresión desde el comienzo de la cadena, luego desde el segundo carácter, y así sucesivamente, hasta que encuentra una coincidencia o llega al final de la cadena. Devolverá la

primera coincidencia que encuentre o fracasará en encontrar cualquier coincidencia.

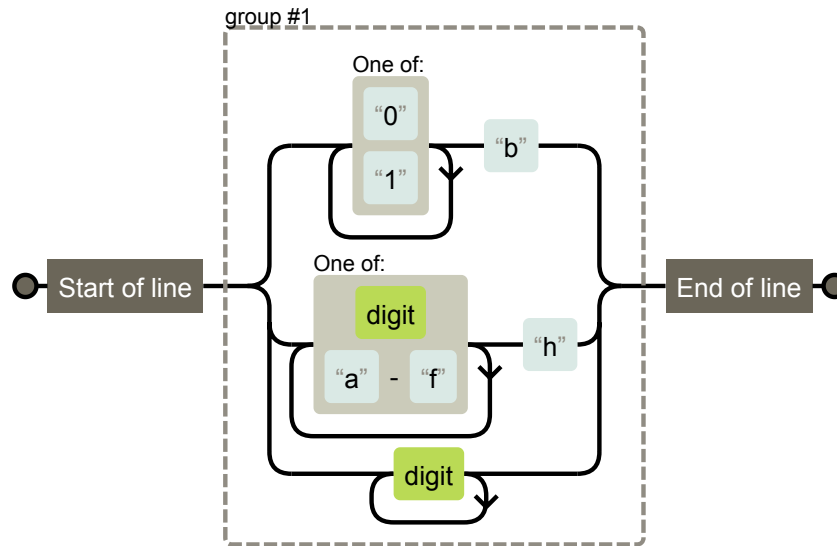
Para hacer la coincidencia real, el motor trata a una expresión regular algo así como un diagrama de flujo. Este es el diagrama para la expresión de ganado en el ejemplo anterior:



Nuestra expresión coincide si podemos encontrar un camino desde el lado izquierdo del diagrama hasta el lado derecho. Mantenemos una posición actual en la cadena, y cada vez que avanzamos a través de un recuadro, verificamos que la parte de la cadena después de nuestra posición actual coincida con ese recuadro.

RETROCESO

La expresión regular `/^([01]+b|[\da-f]+h|\d+)$/` coincide ya sea con un número binario seguido de una *b*, un número hexadecimal (es decir, base 16, con las letras *a* a *f* representando los dígitos del 10 al 15) seguido de un *h*, o un número decimal regular sin un carácter de sufijo. Este es el diagrama correspondiente:



Al coincidir con esta expresión, a menudo sucede que se ingresa por la rama superior (binaria) aunque la entrada en realidad no contenga un número binario. Al coincidir con la cadena "103", por ejemplo, solo se aclara en el 3 que estamos en la rama incorrecta. La cadena *coincide* con la expresión, simplemente no con la rama en la que nos encontramos actualmente.

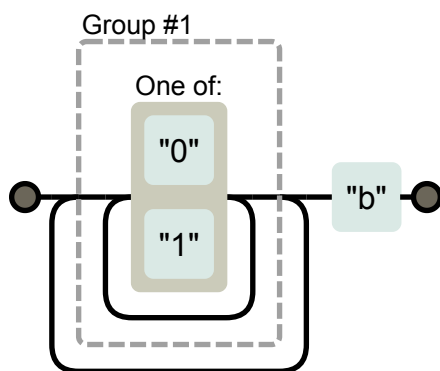
Entonces, el coincidente *retrocede*. Al ingresar a una rama, recuerda su posición actual (en este caso, al principio de la cadena, justo después del primer cuadro de límite en el diagrama) para poder retroceder y probar otra rama si la actual no funciona. Para la cadena "103", después de encontrar el carácter 3, intentará la rama para los números hexadecimales, lo cual también falla porque no hay un *h* después del número. Entonces intenta la rama para los números decimales. Esta encaja, y se informa una coincidencia después de todo.

El coincidente se detiene tan pronto como encuentra una coincidencia completa. Esto significa que si varias ramas podrían

coincidir potencialmente con una cadena, solo se usa la primera (ordenada por dónde aparecen las ramas en la expresión regular).

El retroceso también ocurre para los operadores de repetición como $+$ y $*$. Si coincide con $/^{\wedge} \cdot x/$ contra "abcxe", la parte $\cdot x$ intentará primero consumir toda la cadena. Luego el motor se dará cuenta de que necesita una x para que coincida con el patrón. Dado que no hay una x más allá del final de la cadena, el operador estrella intentará coincidir con un carácter menos. Pero el coincidente no encuentra una x después de $abcx$ tampoco, por lo que retrocede nuevamente, coincidiendo con el operador estrella solo con abc . Ahora encuentra una x donde la necesita y reporta una coincidencia exitosa desde las posiciones 0 a 4.

Es posible escribir expresiones regulares que realizarán *mucho* retroceso. Este problema ocurre cuando un patrón puede coincidir con una parte de la entrada de muchas formas diferentes. Por ejemplo, si nos confundimos al escribir una expresión regular para los números binarios, podríamos escribir accidentalmente algo como $/([01]^+)+b/$.



Si intenta hacer coincidir una serie larga de ceros y unos sin un carácter b al final, el analizador primero pasa por el bucle interno

hasta que se queda sin dígitos. Luego se da cuenta de que no hay *b*, por lo que retrocede una posición, pasa por el bucle externo una vez y vuelve a darse por vencido, intentando retroceder nuevamente fuera del bucle interno. Continuará intentando todas las rutas posibles a través de estos dos bucles. Esto significa que la cantidad de trabajo se *duplica* con cada carácter adicional. Incluso con apenas unas pocas docenas de caracteres, la coincidencia resultante tomará prácticamente para siempre.

EL MÉTODO REPLACE

Los valores de cadena tienen un método `replace` que se puede utilizar para reemplazar parte de la cadena con otra cadena.

```
console.log("papa".replace("p", "m"));  
// → mapa
```

El primer argumento también puede ser una expresión regular, en cuyo caso se reemplaza la primera coincidencia de la expresión regular. Cuando se agrega una opción *g* (para *global*) después de la expresión regular, *todas* las coincidencias en la cadena serán reemplazadas, no solo la primera.

```
console.log("Borobudur".replace(/[ou]/, "a"));  
// → Barobudur  
console.log("Borobudur".replace(/[ou]/g, "a"));  
// → Barabadar
```

El verdadero poder de usar expresiones regulares con `replace` proviene del hecho de que podemos hacer referencia a grupos coincidentes en la cadena de reemplazo. Por ejemplo, digamos que tenemos una cadena larga que contiene los nombres de personas, un

nombre por línea, en el formato Apellido , Nombre. Si queremos intercambiar estos nombres y eliminar la coma para obtener un formato Nombre Apellido, podemos usar el siguiente código:

```
console.log(
  "Liskov, Barbara\nMcCarthy, John\nMilner, Robin"
  .replace(/(\p{L}+)/, (\p{L}+)/gu, "$2 $1"));
// → Barbara Liskov
//   John McCarthy
//   Robin Milner
```

Los \$1 y \$2 en la cadena de reemplazo se refieren a los grupos entre paréntesis en el patrón. \$1 es reemplazado por el texto que coincidió con el primer grupo, \$2 por el segundo, y así sucesivamente, hasta \$9. Toda la coincidencia se puede referenciar con \$&.

Es posible pasar una función, en lugar de una cadena, como segundo argumento a `replace`. Para cada reemplazo, la función se llamará con los grupos coincidentes (así como la coincidencia completa) como argumentos, y su valor de retorno se insertará en la nueva cadena.

Aquí tienes un ejemplo:

```
let stock = "1 limón, 2 repollos y 101 huevos";
function menosUno(match, cantidad, unidad) {
  cantidad = Number(cantidad) - 1;
  if (cantidad == 1) { // solo queda uno, se elimina la 's'
    unidad = unidad.slice(0, unidad.length - 1);
  } else if (cantidad == 0) {
    cantidad = "ningún";
  }
  return cantidad + " " + unidad;
}
console.log(stock.replace(/(\d+) (\p{L}+)/gu, menosUno));
// → ningún limón, 1 repollo y 100 huevos
```

Esta función toma una cadena, encuentra todas las ocurrencias de un número seguido de una palabra alfanumérica, y devuelve una cadena que tiene una cantidad menos de cada una de esas ocurrencias.

El grupo `(\d+)` termina siendo el argumento `amount` de la función, y el grupo `(\p{L}+)` se asigna a `unit`. La función convierte `amount` a un número, lo cual siempre funciona ya que coincide con `\d+`, y realiza algunos ajustes en caso de que solo quede uno o ninguno.

AVARICIA

Es posible usar `replace` para escribir una función que elimine todos los comentarios de un fragmento de código JavaScript. Aquí tienes un primer intento:

```
function stripComments(code) {  
    return code.replace(/\/\/*.*|\/\/*\[^\]*\*\/g, "");  
}  
console.log(stripComments("1 + /* 2 */3"));  
// → 1 + 3  
console.log(stripComments("x = 10; // ¡diez!"));  
// → x = 10;  
console.log(stripComments("1 /* a */+/* b */ 1"));  
// → 1 1
```

La parte antes del operador *or* coincide con dos caracteres de barra seguidos por cualquier cantidad de caracteres que no sean de nueva línea. La parte de comentarios de varias líneas es más compleja. Utilizamos `[^]` (cualquier carácter que no esté en el conjunto vacío de caracteres) como una forma de coincidir con cualquier carácter. No podemos usar simplemente un punto aquí porque los

comentarios de bloque pueden continuar en una nueva línea, y el carácter de punto no coincide con caracteres de nueva línea.

Pero la salida para la última línea parece haber salido mal. ¿Por qué?

La parte `[^]*` de la expresión, como describí en la sección sobre retroceso, primero intentará coincidir con todo lo que pueda. Si esto hace que la siguiente parte del patrón falle, el coincidente retrocede un carácter y vuelve a intentar desde ahí. En el ejemplo, el coincidente intenta primero coincidir con el resto completo de la cadena y luego retrocede desde allí. Encontrará una ocurrencia de `*/` después de retroceder cuatro caracteres y coincidirá con eso. Esto no es lo que queríamos, la intención era coincidir con un único comentario, no llegar hasta el final del código y encontrar el final del último comentario de bloque.

Debido a este comportamiento, decimos que los operadores de repetición (`+`, `*`, `?`, y `{}`) son *avariciosos*, lo que significa que coinciden con todo lo que pueden y retroceden desde allí. Si colocas un signo de interrogación después de ellos (`+`, `*`, `?`, `{}`), se vuelven no avariciosos y comienzan coincidiendo con la menor cantidad posible, coincidiendo más solo cuando el patrón restante no encaja con la coincidencia más pequeña.

Y eso es exactamente lo que queremos en este caso. Al hacer que el asterisco coincida con la menor cantidad de caracteres que nos lleva a `*/`, consumimos un comentario de bloque y nada más.

```
function stripComments(code) {  
    return code.replace(/\\/\\. *|\\/\\*[^]*?\\*\\/g, "");  
}
```

```
console.log(stripComments("1 /* a */+/* b */ 1"));  
// → 1 + 1
```

Muchos errors en programas de expresión regular pueden rastrearse hasta el uso no intencionado de un operador avaricioso donde uno no avaricioso funcionaría mejor. Cuando uses un operador de repetición, prefiere la variante no avariciosa.

CREACIÓN DINÁMICA DE OBJETOS REGEXP

Hay casos en los que es posible que no sepas el patrón exacto que necesitas para hacer coincidir cuando estás escribiendo tu código. Digamos que quieres probar el nombre de usuario en un fragmento de texto. Puedes construir una cadena y usar el constructor `RegExp` en ello. Aquí tienes un ejemplo:

```
let name = "harry";  
let regexp = new RegExp("(^|\\s)" + name + "($|\\s)", "gi");  
console.log(regexp.test("Harry es un personaje dudoso."));  
// → true
```

Al crear la parte `\\s` de la cadena, tenemos que usar dos barras invertidas porque las estamos escribiendo en una cadena normal, no en una expresión regular entre barras. El segundo argumento del constructor `RegExp` contiene las opciones para la expresión regular, en este caso, `"gi"` para global e insensible a mayúsculas y minúsculas.

Pero ¿qué pasa si el nombre es `"dea+hl[]rd"` porque nuestro usuario es un adolescente nerd? Eso resultaría en una expresión regular absurda que en realidad no coincidiría con el nombre del usuario.

Para solucionar esto, podemos agregar barras invertidas antes de cualquier carácter que tenga un significado especial.

```
let name = "dea+hl[]rd";
let escaped = name.replace(/[\\[. +*?(){}|^$]/g, "\\$&");
let regexp = new RegExp("(^|\\s)" + escaped + "(\\s|$)",
                        "gi");
let text = "Este chico dea+hl[]rd es súper molesto.";
console.log(regexp.test(text));
// → true
```

EL MÉTODO SEARCH

El método `indexOf` en las cadenas no puede ser llamado con una expresión regular. Pero hay otro método, `search`, que espera una expresión regular. Al igual que `indexOf`, devuelve el primer índice en el que se encontró la expresión, o -1 cuando no se encontró.

```
console.log(" palabra".search(/S/));
// → 2
console.log(" ".search(/S/));
// → -1
```

Desafortunadamente, no hay una forma de indicar que la coincidencia debería comenzar en un offset dado (como se puede hacer con el segundo argumento de `indexOf`), lo cual a menudo sería útil.

LA PROPIEDAD LASTINDEX

El método `exec` de manera similar no proporciona una forma conveniente de comenzar a buscar desde una posición dada en la cadena. Pero sí proporciona una forma *inconveniente*.

Los objetos de expresión regular tienen propiedades. Una de esas propiedades es `source`, que contiene la cadena de la que se creó la expresión. Otra propiedad es `lastIndex`, que controla, en algunas circunstancias limitadas, desde dónde comenzará la siguiente coincidencia.

Estas circunstancias implican que la expresión regular debe tener la opción global (`g`) o pegajosa (`y`) activada, y la coincidencia debe ocurrir a través del método `exec`. Nuevamente, una solución menos confusa habría sido simplemente permitir que se pase un argumento adicional a `exec`, pero la confusión es una característica esencial de la interfaz de expresiones regulares de JavaScript.

```
let pattern = /y/g;
pattern.lastIndex = 3;
let match = pattern.exec("xyzzzy");
console.log(match.index);
// → 4
console.log(pattern.lastIndex);
// → 5
```

Si la coincidencia tuvo éxito, la llamada a `exec` actualiza automáticamente la propiedad `lastIndex` para que apunte después de la coincidencia. Si no se encontró ninguna coincidencia, `lastIndex` se restablece a cero, que es también el valor que tiene en un objeto de expresión regular recién construido.

La diferencia entre las opciones `global` y `sticky` es que, cuando se habilita `sticky`, la coincidencia solo se producirá si comienza directamente en `lastIndex`, mientras que con `global` se buscará una posición donde pueda comenzar una coincidencia.

```
let global = /abc/g;
console.log(global.exec("xyz abc"));
// → ["abc"]
let sticky = /abc/y;
console.log(sticky.exec("xyz abc"));
// → null
```

Al usar un valor de expresión regular compartido para múltiples llamadas a `exec`, estas actualizaciones automáticas a la propiedad `lastIndex` pueden causar problemas. Es posible que tu expresión regular comience accidentalmente en un índice que quedó de una llamada previa.

```
let digit = /\d/g;
console.log(digit.exec("aquí está: 1"));
// → ["1"]
console.log(digit.exec("ahora: 1"));
// → null
```

Otro efecto interesante de la opción global es que cambia la forma en que funciona el método `match` en las cadenas. Cuando se llama con una expresión global, en lugar de devolver una matriz similar a la devuelta por `exec`, `match` encontrará *todas* las coincidencias del patrón en la cadena y devolverá una matriz que contiene las cadenas coincidentes.

```
console.log("Banana".match(/an/g));
// → ["an", "an"]
```

Así que ten cuidado con las expresiones regulares globales. Los casos en los que son necesarias, como las llamadas a `replace` y los lugares donde quieres usar explícitamente `lastIndex`, son típicamente los únicos lugares donde las desees utilizar.

OBTENIENDO TODAS LAS COINCIDENCIAS

Algo común que se hace es encontrar todas las coincidencias de una expresión regular en una cadena. Podemos hacer esto usando el método `matchAll`.

```
let input = "Una cadena con 3 números... 42 y 88.";
let matches = input.matchAll(/\d+/g);
for (let match of matches) {
  console.log("Encontrado", match[0], "en", match.index);
}
// → Encontrado 3 en 14
//   Encontrado 42 en 33
//   Encontrado 88 en 40
```

Este método devuelve una matriz de matrices de coincidencias. La expresión regular que se le proporciona *debe* tener `g` habilitado.

ANALIZANDO UN ARCHIVO INI

Para concluir el capítulo, analizaremos un problema que requiere expresiones regulares. Imagina que estamos escribiendo un programa para recopilar automáticamente información sobre nuestros enemigos desde Internet. (En realidad, no escribiremos ese programa aquí, solo la parte que lee el archivo de configuración. Lo siento.) El archivo de configuración se ve así:

```
motorbusqueda=https://duckduckgo.com/?q=$1
rencor=9.7

; comentarios precedidos por un punto y coma...
; cada sección se refiere a un enemigo individual
[larry]
fullname=Larry Doe
type=matón de jardín de infantes
website=http://www.geocities.com/CapeCanaveral/11451
```

```
[davaeorn]
fullname=Davaeorn
type=mago malvado
outputdir=/home/marijn/enemies/davaeorn
```

Las reglas exactas para este formato (que es un formato ampliamente utilizado, generalmente llamado un archivo *INI*) son las siguientes:

- Las líneas en blanco y las líneas que comienzan con punto y coma son ignoradas.
- Las líneas envueltas en [y] inician una nueva sección.
- Las líneas que contienen un identificador alfanumérico seguido de un carácter = agregan una configuración a la sección actual.
- Cualquier otra cosa es inválida.

Nuestra tarea es convertir una cadena como esta en un objeto cuyas propiedades contienen cadenas para las configuraciones escritas antes del primer encabezado de sección y subobjetos para las secciones, con esos subobjetos conteniendo las configuraciones de la sección.

Dado que el formato debe procesarse línea por línea, dividir el archivo en líneas separadas es un buen comienzo. Vimos el método `split` en [Capítulo 4](#). Sin embargo, algunos sistemas operativos utilizan no solo un carácter de nueva línea para separar líneas sino un carácter de retorno de carro seguido de una nueva línea (`"\r\n"`). Dado que el método `split` también permite una expresión regular como argumento, podemos usar una expresión

regular como `/\r?\n/` para dividir de una manera que permita tanto `"\n"` como `"\r\n"` entre líneas.

```
function parseINI(string) {
  // Comenzar con un objeto para contener los campos de nivel
  superior
  let result = {};
  let section = result;
  for (let line of string.split(/\r?\n/)) {
    let match;
    if (match = line.match(/^(w+)=(.*)$/)) {
      section[match[1]] = match[2];
    } else if (match = line.match(/^[^(.*)\]$$/)) {
      section = result[match[1]] = {};
    } else if (!/^\s*(;|$)/.test(line)) {
      throw new Error("La línea '" + line + "' no es válida.");
    }
  };
  return result;
}

console.log(parseINI(`
name=Vasilis
[address]
city=Tessaloniki`));
// → {name: "Vasilis", address: {city: "Tessaloniki"}}
```

El código recorre las líneas del archivo y construye un objeto. Las propiedades en la parte superior se almacenan directamente en ese objeto, mientras que las propiedades encontradas en secciones se almacenan en un objeto de sección separado. El enlace `section` apunta al objeto para la sección actual.

Hay dos tipos de líneas significativas: encabezados de sección o líneas de propiedades. Cuando una línea es una propiedad regular, se almacena en la sección actual. Cuando es un encabezado de sección,

se crea un nuevo objeto de sección y `section` se establece para apuntar a él.

Observa el uso recurrente de `^` y `$` para asegurarse de que la expresión coincida con toda la línea, no solo parte de ella. Dejarlos fuera resulta en un código que funciona en su mayor parte pero se comporta de manera extraña para algunas entradas, lo que puede ser un error difícil de rastrear.

``El patrón `if (match = string.match(...))` hace uso del hecho de que el valor de una expresión de asignación (`=`) es el valor asignado. A menudo no estás seguro de que tu llamada a `match` tendrá éxito, por lo que solo puedes acceder al objeto resultante dentro de una declaración `if` que comprueba esto. Para no romper la agradable cadena de formas de `else if`, asignamos el resultado de la coincidencia a un enlace y usamos inmediatamente esa asignación como la prueba para la declaración `if`.

Si una línea no es un encabezado de sección o una propiedad, la función verifica si es un comentario o una línea vacía usando la expresión `/^\s*(;|$)/` para hacer coincidir líneas que solo contienen espacio o espacio seguido de un punto y coma (haciendo que el resto de la línea sea un comentario). Cuando una línea no coincide con ninguna de las formas esperadas, la función lanza una excepción.

UNIDADES DE CÓDIGO Y CARACTERES

Otro error de diseño que se ha estandarizado en las expresiones regulares de JavaScript es que, por defecto, operadores como `.` o `?` trabajan en unidades de código, como se discute en [Capítulo 5](#), no en

caracteres reales. Esto significa que los caracteres que están compuestos por dos unidades de código se comportan de manera extraña.

```
console.log(/🍏{3}/.test("🍏🍏🍏"));  
// → false  
console.log(/<.>/.test("<🌹>"));  
// → false  
console.log(/<.>/u.test("<🌹>"));  
// → true
```

El problema es que el 🍏 en la primera línea se trata como dos unidades de código, y la parte {3} se aplica solo al segundo. Del mismo modo, el punto coincidirá con una sola unidad de código, no con las dos que componen la rosa emoji.

Debes agregar la opción u (Unicode) a tu expresión regular para que trate correctamente este tipo de caracteres.

```
console.log(/🍏{3}/u.test("🍏🍏🍏"));  
// → true
```

RESUMEN

Las expresiones regulares son objetos que representan patrones en cadenas. Utilizan su propio lenguaje para expresar estos patrones.

/abc/	Una secuencia de caracteres
/[abc]/	Cualquier carácter de un conjunto de caracteres
/[^abc]/	Cualquier carácter <i>que no esté</i> en un conjunto de caracteres
/[0-9]/	Cualquier carácter en un rango de caracteres

<code>/x+ /</code>	Una o más ocurrencias del patrón <code>x</code>
<code>/x+? /</code>	Una o más ocurrencias, perezoso
<code>/x* /</code>	Cero o más ocurrencias
<code>/x? /</code>	Cero o una ocurrencia
<code>/x{2,4} /</code>	Dos a cuatro ocurrencias
<code>/(abc) /</code>	Un grupo
<code>/a b c /</code>	Cualquiera de varias combinaciones de patrones
<code>/\d /</code>	Cualquier carácter de dígito
<code>/\w /</code>	Un carácter alfanumérico (“carácter de palabra”)
<code>/\s /</code>	Cualquier carácter de espacio en blanco
<code>/./</code>	Cualquier carácter excepto saltos de línea
<code>/\p{L} /u</code>	Cualquier carácter de letra
<code>/^ /</code>	Inicio de entrada
<code>/\$ /</code>	Fin de entrada
<code>/(?=a) /</code>	Una prueba de vistazo hacia adelante

Una expresión regular tiene un método `test` para comprobar si una cadena dada coincide con ella. También tiene un método `exec` que, cuando se encuentra una coincidencia, devuelve un array que contiene todos los grupos coincidentes. Dicho array tiene una propiedad `index` que indica dónde empezó la coincidencia. Las cadenas tienen un método `match` para compararlas con una expresión regular y un método `search` para buscar una, devolviendo solo la posición de inicio de la coincidencia. Su método `replace` puede reemplazar coincidencias de un patrón con una cadena o función de reemplazo.

Las expresiones regulares pueden tener opciones, que se escriben después de la barra de cierre. La opción `i` hace que la coincidencia no distinga entre mayúsculas y minúsculas. La opción `g` hace que la expresión sea *global*, lo que, entre otras cosas, hace que el método `replaceAll` reemplace todas las instancias en lugar de solo la primera. La opción `y` la hace persistente, lo que significa que no buscará por delante ni omitirá parte de la cadena al buscar una coincidencia. La opción `u` activa el modo Unicode, que habilita la sintaxis `\p` y soluciona varios problemas en torno al manejo de caracteres que ocupan dos unidades de código.

Las expresiones regulares son una herramienta afilada con un mango incómodo. Simplifican enormemente algunas tareas, pero pueden volverse rápidamente ingobernables cuando se aplican a problemas complejos. Parte de saber cómo usarlas es resistir la tentación de intentar forzar cosas que no pueden expresarse de forma clara en ellas.

EJERCICIOS

Es casi inevitable que, al trabajar en estos ejercicios, te sientas confundido y frustrado por el comportamiento inexplicable de algunas expresiones regulares. A veces ayuda introducir tu expresión en una herramienta en línea como debuggex.com para ver si su visualización corresponde a lo que pretendías y para experimentar con la forma en que responde a diferentes cadenas de entrada.

REGEXP GOLF

Code golf es un término utilizado para el juego de intentar expresar un programa en particular con la menor cantidad de caracteres

posible. De manera similar, *regex golf* es la práctica de escribir una expresión regular lo más pequeña posible para que coincida con un patrón dado, y *solo* ese patrón.

Para cada uno de los siguientes elementos, escribe una expresión regular para comprobar si el patrón dado ocurre en una cadena. La expresión regular debe coincidir solo con cadenas que contengan el patrón. Cuando tu expresión funcione, verifica si puedes hacerla más pequeña.

1. *car* y *cat*
2. *pop* y *prop*
3. *ferret*, *ferry* y *ferrari*
4. Cualquier palabra que termine en *iou*s
5. Un carácter de espacio en blanco seguido de un punto, coma, dos puntos o punto y coma
6. Una palabra con más de seis letras
7. Una palabra sin la letra *e* (o *E*)

Consulta la tabla en el [resumen del capítulo](#) para obtener ayuda. Prueba cada solución con algunas cadenas de prueba.

ESTILO DE COMILLAS

Imagina que has escrito una historia y usaste comillas simples single-quote character para marcar piezas de diálogo. Ahora quieres reemplazar todas las comillas de diálogo con comillas dobles, manteniendo las comillas simples utilizadas en contracciones como *aren't*.

Piensa en un patrón que distinga estos dos tipos de uso de comillas y crea una llamada al método `replace` que realice el reemplazo adecuado.

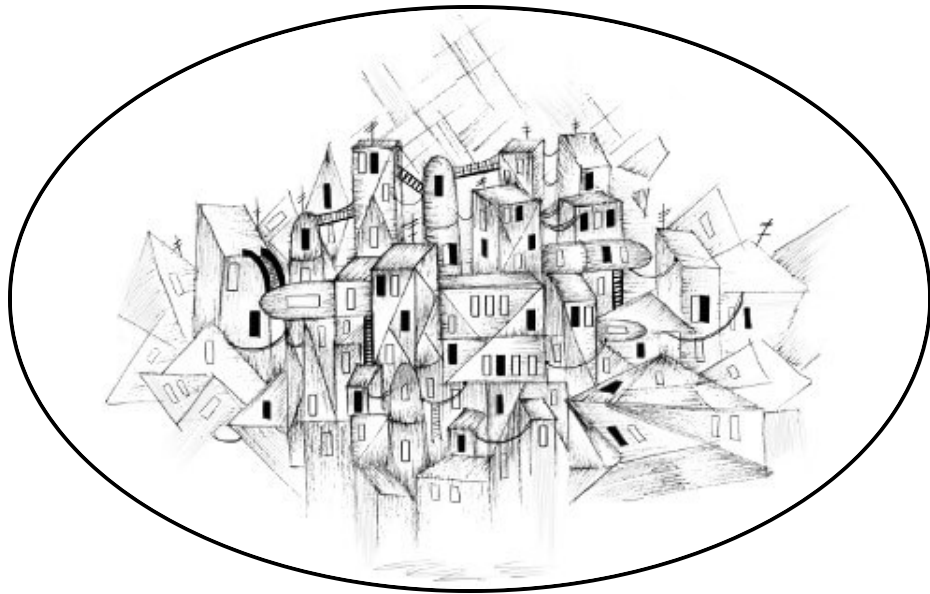
NÚMEROS NUEVAMENTE

Escribe una expresión que coincida solo con los números al estilo de JavaScript. Debe admitir un signo menos o más opcional delante del número, el punto decimal y la notación de exponente— `5e-3` o `1E10`—de nuevo con un signo opcional delante del exponente. También ten en cuenta que no es necesario que haya dígitos delante o después del punto, pero el número no puede ser solo un punto. Es decir, `.5` y `5.` son números de JavaScript válidos, pero un punto solitario *no* lo es.

MÓDULOS

“Escribe código que sea fácil de borrar, no fácil de extender”

—Tef, *La programación es terrible*



Idealmente, un programa tiene una estructura clara y directa. La forma en que funciona es fácil de explicar, y cada parte desempeña un papel bien definido.

En la práctica, los programas crecen de forma orgánica. Se añaden piezas de funcionalidad a medida que el programador identifica nuevas necesidades. Mantener un programa de esta manera bien estructurado requiere atención y trabajo constantes. Este es un trabajo que solo dará sus frutos en el futuro, la próxima vez que alguien trabaje en el programa. Por lo tanto, es tentador descuidarlo

y permitir que las diversas partes del programa se enreden profundamente.

Esto causa dos problemas prácticos. Primero, entender un sistema enredado es difícil. Si todo puede afectar a todo lo demás, es difícil ver cualquier pieza en aislamiento. Te ves obligado a construir una comprensión holística de todo el conjunto. Segundo, si deseas utilizar alguna funcionalidad de dicho programa en otra situación, puede ser más fácil reescribirla que intentar desenredarla de su contexto.

La frase “gran bola de barro” se usa a menudo para tales programas grandes y sin estructura. Todo se une, y al intentar sacar una pieza, todo el conjunto se desintegra y solo logras hacer un desastre.

PROGRAMAS MODULARES

Los *módulos* son un intento de evitar estos problemas. Un módulo es una parte de un programa que especifica en qué otras piezas se basa y qué funcionalidad proporciona para que otros módulos la utilicen (su *interfaz*).

Las interfaces de los módulos tienen mucho en común con las interfaces de objetos, como las vimos en [Capítulo 6](#). Permiten que una parte del módulo esté disponible para el mundo exterior y mantienen el resto privado.

Pero la interfaz que un módulo proporciona para que otros la utilicen es solo la mitad de la historia. Un buen sistema de módulos también requiere que los módulos especifiquen qué código *ellos* utilizan de otros módulos. Estas relaciones se llaman *dependencias*. Si el

módulo A utiliza funcionalidad del módulo B, se dice que *depende* de él. Cuando estas dependencias se especifican claramente en el propio módulo, se pueden utilizar para averiguar qué otros módulos deben estar presentes para poder utilizar un módulo dado y cargar las dependencias automáticamente.

Cuando las formas en que los módulos interactúan entre sí son explícitas, un sistema se vuelve más como LEGO, donde las piezas interactúan a través de conectores bien definidos, y menos como barro, donde todo se mezcla con todo.

MÓDULOS ES

El lenguaje original JavaScript no tenía ningún concepto de un módulo. Todos los scripts se ejecutaban en el mismo ámbito, y acceder a una función definida en otro script se hacía mediante la referencia a las vinculaciones globales creadas por ese script. Esto fomentaba activamente el enredo accidental y difícil de detectar del código e invitaba a problemas como scripts no relacionados que intentaban usar el mismo nombre de vinculación.

Desde ECMAScript 2015, JavaScript admite dos tipos diferentes de programas. Los *scripts* se comportan de la manera antigua: sus vinculaciones se definen en el ámbito global y no tienen forma de referenciar directamente otros scripts. Los *módulos* obtienen su propio ámbito separado y admiten las palabras clave `import` y `export`, que no están disponibles en los scripts, para declarar sus dependencias e interfaz. Este sistema de módulos se suele llamar *módulos de ES* (donde “ES” significa “ECMAScript”).

Un programa modular está compuesto por varios de estos módulos, conectados a través de sus importaciones y exportaciones.

Este ejemplo de módulo convierte entre nombres de días y números (como los devueltos por el método `getDay` de `Date`). Define una constante que no forma parte de su interfaz y dos funciones que sí lo son. No tiene dependencias.

```
const names = ["Domingo", "Lunes", "Martes", "Miércoles",
               "Jueves", "Viernes", "Sábado"];

export function dayName(number) {
  return names[number];
}
export function dayNumber(name) {
  return names.indexOf(name);
}
```

La palabra clave `export` se puede colocar delante de una función, clase o definición de vinculación para indicar que esa vinculación es parte de la interfaz del módulo. Esto permite que otros módulos utilicen esa vinculación importándola.

```
import {dayName} from "../dayname.js";
let ahora = new Date();
console.log(`Hoy es ${dayName(ahora.getDay())}`);
// → Hoy es Lunes
```

La palabra clave `import`, seguida de una lista de nombres de vinculación entre llaves, hace que las vinculaciones de otro módulo estén disponibles en el módulo actual. Los módulos se identifican por cadenas entre comillas.

Cómo se resuelve un nombre de módulo a un programa real difiere según la plataforma. El navegador los trata como direcciones web, mientras que Node.js los resuelve a archivos. Para ejecutar un módulo, se cargan todos los demás módulos en los que depende, y las vinculaciones exportadas se ponen a disposición de los módulos que las importan.

Las declaraciones de importación y exportación no pueden aparecer dentro de funciones, bucles u otros bloques. Se resuelven de inmediato cuando se carga el módulo, independientemente de cómo se ejecute el código en el módulo, y para reflejar esto, deben aparecer solo en el cuerpo del módulo externo.

Así que la interfaz de un módulo consiste en una colección de vinculaciones con nombres, a las cuales tienen acceso otros módulos que dependen de ellas. Las vinculaciones importadas se pueden renombrar para darles un nuevo nombre local utilizando `as` después de su nombre.

```
import {dayName as nomDeJour} from "../dayname.js";
console.log(nomDeJour(3));
// → Miércoles
```

También es posible que un módulo tenga una exportación especial llamada `default`, que a menudo se usa para módulos que solo exportan un único enlace. Para definir una exportación predeterminada, se escribe `export default` antes de una expresión, una declaración de función o una declaración de clase.

```
export default ["Invierno", "Primavera", "Verano", "Otoño"];
```


Este enlace se importa omitiendo las llaves alrededor del nombre de la importación.

```
import nombresEstaciones from "./nombrsestaciones.js";
```

PAQUETES

Una de las ventajas de construir un programa a partir de piezas separadas y poder ejecutar algunas de esas piezas por separado, es que puedes aplicar la misma pieza en diferentes programas.

Pero, ¿cómo se configura esto? Digamos que quiero usar la función `parseINI` de [Capítulo 9](#) en otro programa. Si está claro de qué depende la función (en este caso, nada), puedo simplemente copiar ese módulo en mi nuevo proyecto y usarlo. Pero luego, si encuentro un error en el código, probablemente lo corrija en el programa con el que estoy trabajando en ese momento y olvide corregirlo también en el otro programa.

Una vez que empieces a duplicar código, rápidamente te darás cuenta de que estás perdiendo tiempo y energía moviendo copias y manteniéndolas actualizadas.

Ahí es donde entran los *paquetes*. Un paquete es un fragmento de código que se puede distribuir (copiar e instalar). Puede contener uno o más módulos y tiene información sobre en qué otros paquetes depende. Un paquete también suele venir con documentación que explica qué hace para que las personas que no lo escribieron aún puedan usarlo.

Cuando se encuentra un problema en un paquete o se añade una nueva característica, se actualiza el paquete. Ahora los programas que dependen de él (que también pueden ser paquetes) pueden copiar la nueva versión para obtener las mejoras que se hicieron en el código.

Trabajar de esta manera requiere infraestructura. Necesitamos un lugar para almacenar y encontrar paquetes y una forma conveniente de instalar y actualizarlos. En el mundo de JavaScript, esta infraestructura es provista por NPM (<https://npmjs.org>).

NPM es dos cosas: un servicio en línea donde puedes descargar (y subir) paquetes y un programa (incluido con Node.js) que te ayuda a instalar y gestionarlos.

En el momento de la escritura, hay más de tres millones de paquetes diferentes disponibles en NPM. Una gran parte de ellos son basura, para ser honesto. Pero casi cada paquete de JavaScript útil y disponible públicamente se puede encontrar en NPM. Por ejemplo, un analizador de archivos INI, similar al que construimos en [Capítulo 9](#), está disponible bajo el nombre del paquete `ini`.

[Capítulo 20](#) mostrará cómo instalar tales paquetes localmente usando el programa de línea de comandos `npm`.

Tener paquetes de calidad disponibles para descargar es extremadamente valioso. Significa que a menudo podemos evitar reinventar un programa que 100 personas han escrito antes y obtener una implementación sólida y bien probada con solo presionar algunas teclas.

El software es barato de copiar, por lo que una vez que alguien lo ha escrito, distribuirlo a otras personas es un proceso eficiente. Pero escribirlo en primer lugar es trabajo, y responder a las personas que han encontrado problemas en el código, o que desean proponer nuevas características, es incluso más trabajo.

Por defecto, eres el propietario de los derechos de autor del código que escribes, y otras personas solo pueden usarlo con tu permiso. Pero porque algunas personas son amables y porque publicar buen software puede ayudarte a volverte un poco famoso entre los programadores, muchos paquetes se publican bajo una licencia que permite explícitamente a otras personas usarlo.

La mayoría del código en NPM tiene esta licencia. Algunas licencias requieren que también publiques el código que construyes sobre el paquete bajo la misma licencia. Otros son menos exigentes, simplemente requiriendo que mantengas la licencia con el código al distribuirlo. La comunidad de JavaScript mayormente utiliza este último tipo de licencia. Al usar paquetes de otras personas, asegúrate de estar al tanto de su licencia.

Ahora, en lugar de escribir nuestro propio analizador de archivos INI, podemos usar uno de NPM.

```
import {parse} from "ini";

console.log(parse("x = 10\ny = 20"));
// → {x: "10", y: "20"}
```

MÓDULOS COMMONJS

Antes de 2015, cuando el lenguaje de JavaScript no tenía un sistema de módulos integrado real, las personas ya estaban construyendo sistemas grandes en JavaScript. Para que funcionara, ellos *necesitaban* módulos.

La comunidad diseñó sus propios sistemas de módulos improvisados sobre el lenguaje. Estos utilizan funciones para crear un alcance local para los módulos y objetos regulares para representar interfaces de módulos.

Inicialmente, las personas simplemente envolvían manualmente todo su módulo en una “expresión de función invocada inmediatamente” para crear el alcance del módulo, y asignaban sus objetos de interfaz a una única variable global.

```
const semana = function() {  
  const nombres = ["Domingo", "Lunes", "Martes", "Miércoles",  
    "Jueves", "Viernes", "Sábado"];  
  return {  
    nombre(numero) { return nombres[numero]; },  
    numero(nombre) { return nombres.indexOf(nombre); }  
  };  
}();  
  
console.log(semana.nombre(semana.numero("Domingo")));  
// → Domingo
```

Este estilo de módulos proporciona aislamiento, hasta cierto punto, pero no declara dependencias. En cambio, simplemente coloca su interfaz en el ámbito global y espera que sus dependencias, si las tiene, hagan lo mismo. Esto no es ideal.

Si implementamos nuestro propio cargador de módulos, podemos hacerlo mejor. El enfoque más ampliamente utilizado para los

módulos de JavaScript agregados se llama *Módulos CommonJS*. Node.js lo utilizaba desde el principio (aunque ahora también sabe cómo cargar módulos ES) y es el sistema de módulos utilizado por muchos paquetes en NPM.

Un módulo CommonJS se ve como un script regular, pero tiene acceso a dos enlaces que utiliza para interactuar con otros módulos. El primero es una función llamada `require`. Cuando llamas a esto con el nombre del módulo de tu dependencia, se asegura de que el módulo esté cargado y devuelve su interfaz. El segundo es un objeto llamado `exports`, que es el objeto de interfaz para el módulo. Comienza vacío y agregas propiedades para definir los valores exportados.

Este módulo de ejemplo CommonJS proporciona una función de formateo de fechas. Utiliza dos packages de NPM: `ordinal` para convertir números en strings como "1st" y "2nd", y `date-names` para obtener los nombres en inglés de los días de la semana y los meses. Exporta una única función, `formatDate`, que recibe un objeto `Date` y una cadena `template`.

La cadena de `template` puede contener códigos que indican el formato, como `YYYY` para el año completo y `Do` para el día ordinal del mes. Puede pasársele una cadena como `"MMMM Do YYYY"` para obtener una salida como "22 de noviembre de 2017".

```
const ordinal = require("ordinal");
const {days, months} = require("date-names");

exports.formatDate = function(date, format) {
  return format.replace(/YYYY|M(MMM)?|Do?|dddd/g, tag => {
    if (tag == "YYYY") return date.getFullYear();
```

```

    if (tag == "M") return date.getMonth();
    if (tag == "MMMM") return months[date.getMonth()];
    if (tag == "D") return date.getDate();
    if (tag == "Do") return ordinal(date.getDate());
    if (tag == "dddd") return days[date.getDay()];
  });
};

```

La interfaz de `ordinal` es una única función, mientras que `date-names` exporta un objeto que contiene múltiples cosas: `days` y `months` son arrays de nombres. La técnica de desestructuración es muy conveniente al crear enlaces para las interfaces importadas.

El módulo añade su función de interfaz a `exports` para que los módulos que dependen de él tengan acceso a ella. Podemos usar el módulo de la siguiente manera:

```

const {formatDate} = require("./format-date.js");

console.log(formatDate(new Date(2017, 9, 13),
                        "dddd the Do"));

// → Viernes 13º

```

CommonJS se implementa con un cargador de módulos que, al cargar un módulo, envuelve su código en una función (dándole su propio ámbito local) y pasa los enlaces `require` y `exports` a esa función como argumentos.

Si asumimos que tenemos acceso a una función `readFile` que lee un archivo por su nombre y nos da su contenido, podemos definir una forma simplificada de `require` de la siguiente manera:

```

function require(name) {
  if (!(name in require.cache)) {
    let code = readFile(name);

```

```
    let exports = require.cache[name] = {};  
    let wrapper = Function("require, exports", code);  
    wrapper(require, exports);  
  }  
  return require.cache[name];  
}  
require.cache = Object.create(null);
```

`Function` es una función interna de JavaScript que recibe una lista de argumentos (como una cadena separada por comas) y una cadena que contiene el cuerpo de la función, devolviendo un valor de función con esos argumentos y ese cuerpo. Este es un concepto interesante, ya que permite que un programa cree nuevas partes del programa a partir de datos de cadena, pero también es peligroso, ya que si alguien logra engañar a tu programa para que introduzca una cadena que ellos proporcionan en `Function`, pueden hacer que el programa haga cualquier cosa que quieran.

JavaScript estándar no proporciona una función como `readFile`, pero diferentes entornos de JavaScript, como el navegador y Node.js, proporcionan sus propias formas de acceder a los archivos. El ejemplo simplemente simula que `readFile` existe.

Para evitar cargar el mismo módulo múltiples veces, `require` mantiene una tienda (caché) de módulos ya cargados. Cuando se llama, primero comprueba si el módulo solicitado ha sido cargado y, si no, lo carga. Esto implica leer el código del módulo, envolverlo en una función y llamarlo.

Al definir `require`, `exports` como parámetros para la función de envoltura generada (y pasar los valores apropiados al llamarla), el

cargador se asegura de que estos enlaces estén disponibles en el ámbito del módulo.

Una diferencia importante entre este sistema y los módulos ES es que las importaciones de módulos ES suceden antes de que comience a ejecutarse el script de un módulo, mientras que `require` es una función normal, invocada cuando el módulo ya está en ejecución. A diferencia de las declaraciones `import`, las llamadas a `require` pueden aparecer dentro de funciones, y el nombre de la dependencia puede ser cualquier expresión que se evalúe a una cadena, mientras que `import` solo permite cadenas simples entre comillas.

La transición de la comunidad de JavaScript desde el estilo CommonJS a los módulos ES ha sido lenta y algo complicada. Pero afortunadamente, ahora estamos en un punto en el que la mayoría de los paquetes populares en NPM proporcionan su código como módulos ES, y Node.js permite que los módulos ES importen desde módulos CommonJS. Por lo tanto, si bien el código CommonJS es algo con lo que te encontrarás, ya no hay una razón real para escribir nuevos programas en este estilo.

COMPILACIÓN Y EMPAQUETADO

Muchos paquetes de JavaScript no están, técnicamente, escritos en JavaScript. Hay extensiones, como TypeScript, el dialecto de verificación de tipos mencionado en el [Capítulo 8](#), que se utilizan ampliamente. A menudo, las personas también comienzan a usar extensiones planeadas para el lenguaje mucho antes de que se agreguen a las plataformas que realmente ejecutan JavaScript.

Para hacer esto posible, *compilan* su código, traducéndolo desde su dialecto de JavaScript elegido a JavaScript antiguo, e incluso a una versión anterior de JavaScript, para que los navegadores puedan ejecutarlo.

Incluir un programa modular que consta de 200 archivos diferentes en una página web produce sus propios problemas. Si recuperar un solo archivo a través de la red lleva 50 milisegundos, cargar todo el programa lleva 10 segundos, o quizás la mitad de eso si puedes cargar varios archivos simultáneamente. Eso es mucho tiempo desperdiciado. Como recuperar un solo archivo grande tiende a ser más rápido que recuperar muchos archivos pequeños, los programadores web han comenzado a usar herramientas que combinan sus programas (que dividieron minuciosamente en módulos) en un solo archivo grande antes de publicarlo en la Web. Estas herramientas se llaman *bundlers*.

Y podemos ir más allá. Aparte del número de archivos, el *tamaño* de los archivos también determina qué tan rápido pueden ser transferidos a través de la red. Por lo tanto, la comunidad de JavaScript ha inventado *minificadores*. Estas son herramientas que toman un programa de JavaScript y lo hacen más pequeño al eliminar automáticamente comentarios y espacios en blanco, renombrar enlaces y reemplazar fragmentos de código con código equivalente que ocupa menos espacio.

Por lo tanto, no es raro que el código que encuentres en un paquete de NPM o que se ejecute en una página web haya pasado por *múltiples* etapas de transformación, convirtiéndose desde JavaScript moderno a JavaScript histórico, luego combinando los módulos en

un solo archivo, y minimizando el código. No entraremos en detalles sobre estas herramientas en este libro ya que hay muchas de ellas, y cuál es popular cambia regularmente. Simplemente ten en cuenta que tales cosas existen, y búscalas cuando las necesites.

DISEÑO DE MÓDULOS

Estructurar programas es uno de los aspectos más sutiles de la programación. Cualquier funcionalidad no trivial puede ser organizada de diversas formas.

Un buen diseño de programa es subjetivo—hay compensaciones implicadas y cuestiones de gusto. La mejor manera de aprender el valor de un diseño bien estructurado es leer o trabajar en muchos programas y notar qué funciona y qué no. No asumas que un desorden doloroso es “simplemente así”. Puedes mejorar la estructura de casi todo pensando más detenidamente en ello.

Un aspecto del diseño de módulos es la facilidad de uso. Si estás diseñando algo que se supone será utilizado por varias personas—o incluso por ti mismo, dentro de tres meses cuando ya no recuerdes los detalles de lo que hiciste—es útil que tu interfaz sea simple y predecible.

Eso puede significar seguir convenciones existentes. Un buen ejemplo es el paquete `ini`. Este módulo imita el objeto estándar JSON al proporcionar funciones `parse` y `stringify` (para escribir un archivo INI), y, como JSON, convierte entre cadenas y objetos simples. Por lo tanto, la interfaz es pequeña y familiar, y después de haber trabajado con ella una vez, es probable que recuerdes cómo usarla.

Incluso si no hay una función estándar o paquete ampliamente utilizado para imitar, puedes mantener tus módulos predecibles utilizando estructuras de datos simples y haciendo una sola cosa enfocada. Muchos de los módulos de análisis de archivos INI en NPM proporcionan una función que lee directamente dicho archivo desde el disco duro y lo analiza, por ejemplo. Esto hace imposible usar dichos módulos en el navegador, donde no tenemos acceso directo al sistema de archivos, y añade complejidad que hubiera sido mejor abordada *componiendo* el módulo con alguna función de lectura de archivos.

Esto señala otro aspecto útil del diseño de módulos—la facilidad con la que algo puede ser compuesto con otro código. Los módulos enfocados en calcular valores son aplicables en una gama más amplia de programas que los módulos más grandes que realizan acciones complicadas con efectos secundarios. Un lector de archivos INI que insiste en leer el archivo desde el disco es inútil en un escenario donde el contenido del archivo proviene de otra fuente.

Relacionado con esto, a veces los objetos con estado son útiles o incluso necesarios, pero si algo se puede hacer con una función, utiliza una función. Varios de los lectores de archivos INI en NPM proporcionan un estilo de interfaz que requiere que primero crees un objeto, luego cargues el archivo en tu objeto, y finalmente uses métodos especializados para acceder a los resultados. Este tipo de enfoque es común en la tradición orientada a objetos, y es terrible. En lugar de hacer una sola llamada a función y continuar, debes realizar el ritual de mover tu objeto a través de sus diversos estados. Y debido a que los datos están envueltos en un tipo de objeto

especializado, todo el código que interactúa con él debe conocer ese tipo, creando interdependencias innecesarias.

A menudo, no se puede evitar definir nuevas estructuras de datos, ya que el estándar del lenguaje proporciona solo algunas básicas, y muchos tipos de datos deben ser más complejos que un array o un mapa. Pero cuando un array es suficiente, utiliza un array.

Un ejemplo de una estructura de datos ligeramente más compleja es el grafo de [Capítulo 7](#). No hay una forma única obvia de representar un grafo en JavaScript. En ese capítulo, utilizamos un objeto cuyas propiedades contienen arrays de strings: los otros nodos alcanzables desde ese nodo.

Existen varios paquetes de búsqueda de rutas en NPM, pero ninguno de ellos utiliza este formato de grafo. Por lo general, permiten que las aristas del grafo tengan un peso, que es el costo o la distancia asociada a ellas. Eso no es posible en nuestra representación.

Por ejemplo, está el paquete `dijkstra.js`. Un enfoque conocido para la búsqueda de rutas, bastante similar a nuestra función `findRoute`, se llama *algoritmo de Dijkstra*, en honor a Edsger Dijkstra, quien lo escribió por primera vez. A menudo se agrega el sufijo `js` a los nombres de los paquetes para indicar que están escritos en JavaScript. Este paquete `dijkstra.js` utiliza un formato de grafo similar al nuestro, pero en lugar de arrays, utiliza objetos cuyos valores de propiedad son números, los pesos de las aristas.

Por lo tanto, si quisiéramos usar ese paquete, deberíamos asegurarnos de que nuestro grafo esté almacenado en el formato que espera. Todas las aristas tienen el mismo peso, ya que nuestro

modelo simplificado trata cada camino como teniendo el mismo coste (una vuelta).

```
const {find_path} = require("dijkstra.js");

let graph = {};
for (let node of Object.keys(roadGraph)) {
  let edges = graph[node] = {};
  for (let dest of roadGraph[node]) {
    edges[dest] = 1;
  }
}

console.log(find_path(graph, "Oficina de Correos", "Cabaña"));
// → ["Oficina de Correos", "Casa de Alicia", "Cabaña"]
```

Esto puede ser una barrera para la composición: cuando varios paquetes están utilizando diferentes estructuras de datos para describir cosas similares, combinarlos es difícil. Por lo tanto, si deseas diseñar para la composabilidad, averigua qué estructuras de datos están utilizando otras personas y, cuando sea posible, sigue su ejemplo.

Diseñar una estructura de módulo adecuada para un programa puede ser difícil. En la fase en la que aún estás explorando el problema, probando diferentes cosas para ver qué funciona, es posible que no quieras preocuparte demasiado por esto, ya que mantener todo organizado puede ser una gran distracción. Una vez que tengas algo que se sienta sólido, es un buen momento para dar un paso atrás y organizarlo.

RESUMEN

Los módulos proporcionan estructura a programas más grandes al separar el código en piezas con interfaces claras y dependencias. La interfaz es la parte del módulo que es visible para otros módulos, y las dependencias son los otros módulos que se utilizan.

Dado que JavaScript históricamente no proporcionaba un sistema de módulos, se construyó el sistema CommonJS sobre él. Luego, en algún momento *obtuvo* un sistema incorporado, que ahora coexiste incómodamente con el sistema CommonJS.

Un paquete es un fragmento de código que se puede distribuir por sí solo. NPM es un repositorio de paquetes de JavaScript. Puedes descargar todo tipo de paquetes útiles (y inútiles) desde aquí.

EJERCICIOS

UN ROBOT MODULAR

Estos son los enlaces que crea el proyecto del [Capítulo 7](#):

```
roads
buildGraph
roadGraph
VillageState
runRobot
randomPick
randomRobot
mailRoute
routeRobot
findRoute
goalOrientedRobot
```

Si tuvieras que escribir ese proyecto como un programa modular, ¿qué módulos crearías? ¿Qué módulo dependería de qué otro

módulo y cómo serían sus interfaces?

¿Qué piezas es probable que estén disponibles preescritas en NPM?

¿Preferirías usar un paquete de NPM o escribirlos tú mismo?

MÓDULO DE CAMINOS

Escribe un módulo ES, basado en el ejemplo del [Capítulo 7](#), que contenga el array de caminos y exporte la estructura de datos de gráfico que los representa como `roadGraph`. Debería depender de un módulo `./graph.js`, que exporta una función `buildGraph` que se utiliza para construir el gráfico. Esta función espera un array de arrays de dos elementos (los puntos de inicio y fin de los caminos).

DEPENDENCIAS CIRCULARES

Una dependencia circular es una situación en la que el módulo A depende de B, y B también, directa o indirectamente, depende de A. Muchos sistemas de módulos simplemente prohíben esto porque, sin importar el orden que elijas para cargar dichos módulos, no puedes asegurarte de que las dependencias de cada módulo se hayan cargado antes de que se ejecute.

Los módulos CommonJS permiten una forma limitada de dependencias cíclicas. Siempre y cuando los módulos no accedan a la interfaz de cada uno hasta después de que terminen de cargarse, las dependencias cíclicas están bien.

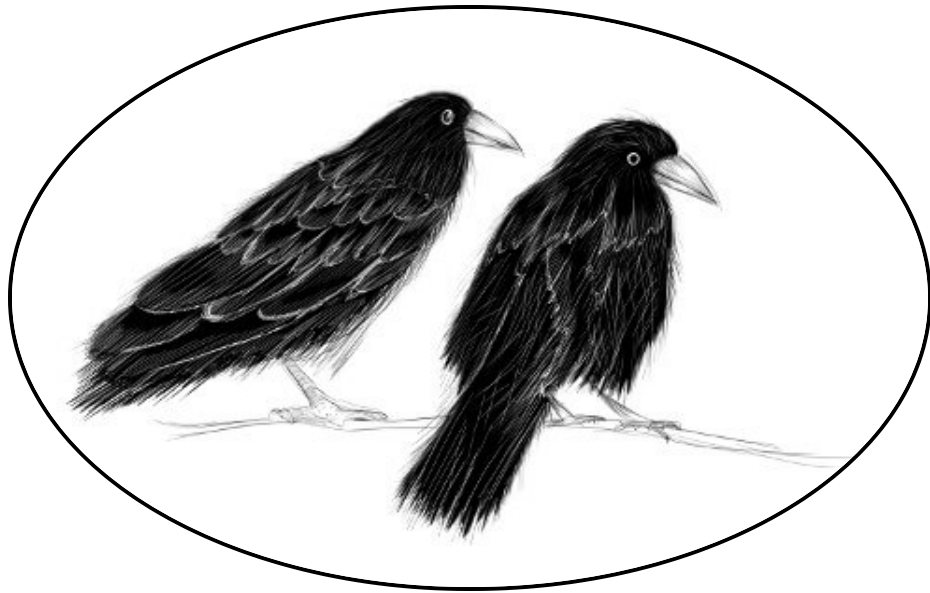
La función `require` proporcionada [anteriormente en este capítulo](#) admite este tipo de ciclo de dependencia. ¿Puedes ver cómo maneja

los ciclos?

PROGRAMACIÓN ASÍNCRONA

“¿Quién puede esperar en silencio mientras el barro se asienta?
¿Quién puede permanecer quieto hasta el momento de la acción?”

—Laozi, *Tao Te Ching*



La parte central de una computadora, la parte que lleva a cabo los pasos individuales que componen nuestros programas, se llama el *procesador*. Los programas que hemos visto hasta ahora mantendrán ocupado al procesador hasta que hayan terminado su trabajo. La velocidad a la cual algo como un bucle que manipula números puede ser ejecutado depende casi enteramente de la velocidad del procesador y la memoria de la computadora.

Pero muchos programas interactúan con cosas fuera del procesador. Por ejemplo, pueden comunicarse a través de una red de computadoras o solicitar datos desde el disco duro, lo cual es mucho más lento que obtenerlo de la memoria.

Cuando esto está sucediendo, sería una lástima dejar el procesador inactivo, ya que podría haber otro trabajo que podría hacer en ese tiempo. En parte, esto es manejado por tu sistema operativo, el cual cambiará el procesador entre múltiples programas en ejecución. Pero eso no ayuda cuando queremos que un *único* programa pueda avanzar mientras espera una solicitud de red.

ASINCRONÍA

En un modelo de programación *sincrónico*, las cosas suceden una a la vez. Cuando llamas a una función que realiza una acción de larga duración, solo devuelve cuando la acción ha terminado y puede devolver el resultado. Esto detiene tu programa durante el tiempo que tome la acción.

Un modelo *asincrónico* permite que múltiples cosas sucedan al mismo tiempo. Cuando inicias una acción, tu programa continúa ejecutándose. Cuando la acción termina, el programa es informado y obtiene acceso al resultado (por ejemplo, los datos leídos desde el disco).

Podemos comparar la programación sincrónica y asincrónica usando un pequeño ejemplo: un programa que realiza dos solicitudes a través de la red y luego combina los resultados.

En un entorno sincrónico, donde la función de solicitud devuelve solo después de haber hecho su trabajo, la forma más fácil de realizar esta tarea es hacer las solicitudes una después de la otra. Esto tiene la desventaja de que la segunda solicitud se iniciará solo cuando la primera haya terminado. El tiempo total tomado será al menos la suma de los dos tiempos de respuesta.

La solución a este problema, en un sistema sincrónico, es iniciar hebras de control adicionales. Una *hebra* es otro programa en ejecución cuya ejecución puede ser intercalada con otros programas por el sistema operativo, ya que la mayoría de las computadoras modernas contienen múltiples procesadores, múltiples hebras incluso podrían ejecutarse al mismo tiempo, en diferentes procesadores. Una segunda hebra podría iniciar la segunda solicitud, y luego ambas hebras esperan que sus resultados regresen, después de lo cual se resincronizan para combinar sus resultados.

En el siguiente diagrama, las líneas gruesas representan el tiempo que el programa pasa funcionando normalmente, y las líneas delgadas representan el tiempo gastado esperando a la red. En el modelo síncrono, el tiempo tomado por la red es *parte* de la línea de tiempo para un hilo de control dado. En el modelo asíncrono, iniciar una acción en la red permite que el programa continúe ejecutándose mientras la comunicación en la red sucede junto a él, notificando al programa cuando haya terminado.

synchronous, single thread of control



synchronous, two threads of control



asynchronous



Otra forma de describir la diferencia es que esperar a que las acciones terminen es *implícito* en el modelo síncrono, mientras que es *explícito*, bajo nuestro control, en el modelo asíncrono.

La asincronía tiene sus pros y sus contras. Facilita la expresión de programas que no encajan en el modelo de control de línea recta, pero también puede hacer que expresar programas que siguen una línea recta sea más complicado. Veremos algunas formas de reducir esta dificultad más adelante en el capítulo.

Tanto las plataformas de programación de JavaScript prominentes — navegadores como Node.js— hacen operaciones que podrían tardar un tiempo de forma asíncrona, en lugar de depender de hilos. Dado que programar con hilos es notoriamente difícil (entender lo que hace un programa es mucho más difícil cuando está haciendo múltiples cosas a la vez), esto generalmente se considera algo bueno.

RETROLLAMADAS

Un enfoque para la programación asíncrona es hacer que las funciones que necesitan esperar por algo tomen un argumento adicional, una *función de devolución de llamada*. La función asíncrona inicia algún proceso, configura las cosas para que se llame

a la función de devolución de llamada cuando el proceso termine, y luego retorna.

Como ejemplo, la función `setTimeout`, disponible tanto en Node.js como en los navegadores, espera un número dado de milisegundos (un segundo equivale a mil milisegundos) y luego llama a una función.

```
setTimeout(() => console.log("Tick"), 500);
```

Esperar no suele ser un tipo de trabajo muy importante, pero puede ser muy útil cuando necesitas organizar que algo suceda en un momento determinado o verificar si alguna otra acción está tomando más tiempo del esperado.

Otro ejemplo de una operación asíncronica común es leer un archivo desde el almacenamiento de un dispositivo. Imagina que tienes una función `readTextFile`, la cual lee el contenido de un archivo como una cadena y lo pasa a una función de devolución de llamada.

```
readTextFile("lista_de_compras.txt", contenido => {  
  console.log(`Lista de Compras:\n${contenido}`);  
});  
// → Lista de Compras:  
// → Mantequilla de cacahuete  
// → Plátanos
```

La función `readTextFile` no es parte del estándar de JavaScript. Veremos cómo leer archivos en el navegador y en Node.js en capítulos posteriores.

Realizar múltiples acciones asíncronas en fila usando devoluciones de llamada significa que tienes que seguir pasando nuevas funciones

para manejar la continuación de la computación después de las acciones. Así es como podría verse una función asíncronica que compara dos archivos y produce un booleano que indica si su contenido es el mismo.

```
function compararArchivos(archivoA, archivoB, devolucionLlamada) {  
  readTextFile(archivoA, contenidoA => {  
    readTextFile(archivoB, contenidoB => {  
      devolucionLlamada(contenidoA == contenidoB);  
    });  
  });  
}
```

Este estilo de programación es funcional, pero el nivel de indentación aumenta con cada acción asíncronica porque terminas en otra función. Hacer cosas más complicadas, como envolver acciones asíncronicas en un bucle, puede ser incómodo.

De alguna manera, la asincronía es contagiosa. Cualquier función que llame a una función que trabaja de forma asíncronica debe ser asíncronica en sí misma, utilizando una devolución de llamada u otro mecanismo similar para entregar su resultado. Llamar a una devolución de llamada es algo más complicado y propenso a errores que simplemente devolver un valor, por lo que necesitar estructurar grandes partes de tu programa de esa manera no es ideal.

PROMESAS

Una forma ligeramente diferente de construir un programa asíncronico es hacer que las funciones asíncronicas devuelvan un objeto que represente su resultado (futuro) en lugar de pasar devoluciones de llamada por todas partes. De esta manera, tales

funciones realmente devuelven algo significativo, y la estructura del programa se asemeja más a la de los programas síncronos.

Para esto sirve la clase estándar `Promise`. Una *promesa* es un recibo que representa un valor que aún puede no estar disponible. Proporciona un método `then` que te permite registrar una función que debe ser llamada cuando la acción por la que está esperando finalice. Cuando la promesa se *resuelve*, es decir, su valor se vuelve disponible, esas funciones (puede haber varias) son llamadas con el valor del resultado. Es posible llamar a `then` en una promesa que ya ha sido resuelta; tu función seguirá siendo llamada.

La forma más sencilla de crear una promesa es llamando a `Promise.resolve`. Esta función se asegura de que el valor que le proporcionas esté envuelto en una promesa. Si ya es una promesa, simplemente se devuelve; de lo contrario, obtienes una nueva promesa que se resuelve de inmediato con tu valor como resultado.

```
let quince = Promise.resolve(15);
quince.then(valor => console.log(`Obtenido ${valor}`));
// → Obtenido 15
```

Para crear una promesa que no se resuelva inmediatamente, puedes utilizar `Promise` como constructor. Tiene una interfaz un tanto extraña: el constructor espera una función como argumento, la cual llama inmediatamente, pasándole una función que puede utilizar para resolver la promesa.

Así es como podrías crear una interfaz basada en promesas para la función `readTextFile`:

```
function textFile(nombreArchivo) {  
  return new Promise(resolve => {  
    readTextFile(nombreArchivo, texto => resolve(texto));  
  });  
}  
  
textFile("planes.txt").then(console.log);
```

Observa cómo esta función asíncrona devuelve un valor significativo: una promesa para proporcionarte el contenido del archivo en algún momento futuro.

Una característica útil del método `then` es que él mismo devuelve otra promesa que se resuelve al valor retornado por la función de devolución de llamada o, si esa función devuelve una promesa, al valor al que esa promesa se resuelve. De esta forma, puedes “encadenar” varias llamadas a `then` para configurar una secuencia de acciones asíncronas.

Esta función, la cual lee un archivo lleno de nombres de archivos y devuelve el contenido de un archivo aleatorio de esa lista, muestra este tipo de cadena asíncrona de promesas.

```
function randomFile(archivoLista) {  
  return textFile(archivoLista)  
    .then(contenido => contenido.trim().split("\n"))  
    .then(ls => ls[Math.floor(Math.random() * ls.length)])  
    .then(nombreArchivo => textFile(nombreArchivo));  
}
```

La función devuelve el resultado de esta cadena de llamadas a `then`. La promesa inicial obtiene la lista de archivos como una cadena. La primera llamada a `then` transforma esa cadena en un array de líneas, produciendo una nueva promesa. La segunda llamada a `then`

elige una línea aleatoria de eso, produciendo una tercera promesa que arroja un único nombre de archivo. La llamada final a `then` lee este archivo, de modo que el resultado de la función en su totalidad es una promesa que devuelve el contenido de un archivo aleatorio.

En este código, las funciones utilizadas en las primeras dos llamadas a `then` devuelven un valor regular, que se pasará inmediatamente a la promesa devuelta por `then` cuando la función regrese. La última devuelve una promesa (`textFile(nombreArchivo)`), convirtiéndola en un paso asíncronico real.

También habría sido posible realizar todos estos pasos dentro de un solo callback de `then`, ya que solo el último paso es realmente asíncrono. Pero los tipos de envolturas `then` que solo realizan alguna transformación de datos síncrona son a menudo útiles, por ejemplo, cuando deseas devolver una promesa que produzca una versión procesada de algún resultado asíncrono.

```
function jsonFile(nombreArchivo) {  
    return textFile(nombreArchivo).then(JSON.parse);  
}  
  
jsonFile("package.json").then(console.log);
```

En general, es útil pensar en las promesas como un mecanismo que permite al código ignorar la pregunta de cuándo va a llegar un valor. Un valor normal tiene que existir realmente antes de que podamos hacer referencia a él. Un valor prometido es un valor que *puede* estar allí o podría aparecer en algún momento en el futuro. Las operaciones definidas en términos de promesas, al conectarlas con llamadas `then`, se ejecutan de forma asíncrona a medida que sus entradas están disponibles.

FALLA

Las computaciones regulares de JavaScript pueden fallar al lanzar una excepción. Las computaciones asíncronas a menudo necesitan algo así. Una solicitud de red puede fallar, un archivo puede no existir, o algún código que forma parte de la computación asíncrona puede lanzar una excepción.

Uno de los problemas más apremiantes con el estilo de programación asíncrona basado en devoluciones de llamada es que hace extremadamente difícil asegurarse de que las fallas se informen adecuadamente a las devoluciones de llamada.

Una convención ampliamente utilizada es que el primer argumento de la devolución de llamada se utiliza para indicar que la acción falló, y el segundo contiene el valor producido por la acción cuando fue exitosa.

```
unaFuncionAsincrona((error, valor) => {  
  if (error) manejarError(error);  
  else procesarValor(valor);  
});
```

Tales funciones de devolución de llamada siempre deben verificar si recibieron una excepción y asegurarse de que cualquier problema que causen, incluidas las excepciones lanzadas por las funciones que llaman, se capturen y se den a la función correcta.

Las promesas facilitan esto. Pueden ser o bien resueltas (la acción se completó con éxito) o rechazadas (falló). Los manejadores de resolución (como se registran con `then`) se llaman solo cuando la acción es exitosa, y los rechazos se propagan a la nueva promesa que

es devuelta por `then`. Cuando un manejador lanza una excepción, esto causa automáticamente que la promesa producida por la llamada a su `then` sea rechazada. Entonces, si algún elemento en una cadena de acciones asíncronas falla, el resultado de toda la cadena se marca como rechazado, y no se llaman manejadores de éxito más allá del punto donde falló.

Al igual que resolver una promesa proporciona un valor, rechazar una también lo hace, generalmente llamado el *motivo* del rechazo. Cuando una excepción en una función manejadora causa el rechazo, el valor de la excepción se usa como el motivo. De manera similar, cuando una función manejadora devuelve una promesa que es rechazada, ese rechazo fluye hacia la siguiente promesa. Existe una función `Promise.reject` que crea una nueva promesa inmediatamente rechazada.

Para manejar explícitamente tales rechazos, las promesas tienen un método `catch` que registra un manejador para ser llamado cuando la promesa es rechazada, similar a cómo los manejadores de `then` manejan la resolución normal. También es muy similar a `then` en que devuelve una nueva promesa, que se resuelve con el valor de la promesa original cuando se resuelve normalmente y con el resultado del manejador `catch` en caso contrario. Si un manejador de `catch` lanza un error, la nueva promesa también se rechaza.

Como un atajo, `then` también acepta un manejador de rechazo como segundo argumento, para poder instalar ambos tipos de manejadores en una sola llamada de método.

Una función pasada al constructor `Promise` recibe un segundo argumento, junto con la función de resolución, que puede usar para

rechazar la nueva promesa. Cuando nuestra función `readTextFile` encuentra un problema, pasa el error a su función de devolución de llamada como segundo argumento. Nuestro envoltorio `textFile` debería realmente examinar ese argumento, de manera que un fallo cause que la promesa que devuelve sea rechazada.

```
function textFile(filename) {
  return new Promise((resolve, reject) => {
    readTextFile(filename, (text, error) => {
      if (error) reject(error);
      else resolve(text);
    });
  });
}
```

Las cadenas de valores de promesa creadas por llamadas a `then` y `catch` forman así un pipeline a través del cual se mueven los valores asíncronos o fallos. Dado que dichas cadenas se crean registrando manejadores, cada eslabón tiene asociado un manejador de éxito o un manejador de rechazo (o ambos). Los manejadores que no coinciden con el tipo de resultado (éxito o fallo) son ignorados. Pero aquellos que coinciden son llamados, y su resultado determina qué tipo de valor viene a continuación: éxito cuando devuelve un valor que no es una promesa, rechazo cuando genera una excepción, y el resultado de la promesa cuando devuelve una promesa.

```
new Promise((_, reject) => reject(new Error("Fail")))
  .then(value => console.log("Manejador 1:", value))
  .catch(reason => {
    console.log("Error capturado " + reason);
    return "nada";
  })
  .then(value => console.log("Manejador 2:", value));
// → Error capturado Error: Fail
// → Handler 2: nothing
```

La primera función de manejador regular no es llamada, porque en ese punto del pipeline la promesa contiene un rechazo. El manejador catch maneja ese rechazo y devuelve un valor, que se le da a la segunda función de manejador.

Cuando una excepción no controlada es manejada por el entorno, los entornos de JavaScript pueden detectar cuándo un rechazo de promesa no es manejado y lo reportarán como un error.

CARLA

Es un día soleado en Berlín. La pista del antiguo aeropuerto desmantelado rebosa de ciclistas y patinadores en línea. En el césped cerca de un contenedor de basura un grupo de cuervos se agita ruidosamente, intentando convencer a un grupo de turistas de que les den sus sándwiches.

Uno de los cuervos destaca: una hembra grande andrajosa con algunas plumas blancas en su ala derecha. Está atrayendo a la gente con habilidad y confianza que sugieren que ha estado haciendo esto durante mucho tiempo. Cuando un anciano se distrae con las travesuras de otro cuervo, ella se abalanza casualmente, arrebatándole su bollo a medio comer de su mano y se aleja planeando.

A diferencia del resto del grupo, que parece estar feliz de pasar el día holgazaneando aquí, el cuervo grande parece tener un propósito. Llevando su botín, vuela directamente hacia el techo del edificio del hangar, desapareciendo en una rejilla de ventilación.

Dentro del edificio, se puede escuchar un sonido peculiar: suave, pero persistente. Viene de un espacio estrecho bajo el techo de una

escalera sin terminar. El cuervo está sentado allí, rodeado de sus botines robados, media docena de teléfonos inteligentes (varios de los cuales están encendidos) y un enredo de cables. Golpea rápidamente la pantalla de uno de los teléfonos con su pico. Aparecen palabras en él. Si no supieras mejor, pensarías que estaba escribiendo. Este cuervo es conocido por sus pares como “cāāw-krö”. Pero dado que esos sonidos no son adecuados para las cuerdas vocales humanas, la llamaremos Carla.

Carla es un cuervo algo peculiar. En su juventud, estaba fascinada por el lenguaje humano, escuchando a la gente hasta que tuvo un buen entendimiento de lo que decían. Más tarde, su interés se trasladó a la tecnología humana, y comenzó a robar teléfonos para estudiarlos. Su proyecto actual es aprender a programar. El texto que está escribiendo en su laboratorio secreto, de hecho, es un fragmento de código JavaScript.

INFILTRACIÓN

A Carla le encanta Internet. Fastidiosamente, el teléfono en el que está trabajando está a punto de quedarse sin datos prepagos. El edificio tiene una red inalámbrica, pero se requiere un código para acceder a ella.

Afortunadamente, los enrutadores inalámbricos en el edificio tienen 20 años y están mal protegidos. Tras investigar un poco, Carla descubre que el mecanismo de autenticación de la red tiene una falla que puede aprovechar. Al unirse a la red, un dispositivo debe enviar el código correcto de 6 dígitos. El punto de acceso responderá con un mensaje de éxito o fracaso dependiendo de si se proporciona el

código correcto. Sin embargo, al enviar solo un código parcial (digamos, solo 3 dígitos), la respuesta es diferente según si esos dígitos son el inicio correcto del código o no. Cuando se envía un número incorrecto, se recibe inmediatamente un mensaje de fracaso. Cuando se envían los correctos, el punto de acceso espera más dígitos.

Esto hace posible acelerar enormemente la adivinación del número. Carla puede encontrar el primer dígito probando cada número a su vez, hasta que encuentre uno que no devuelva inmediatamente un fracaso. Teniendo un dígito, puede encontrar el segundo de la misma manera, y así sucesivamente, hasta que conozca todo el código de acceso.

Supongamos que tenemos una función `joinWifi`. Dado el nombre de la red y el código de acceso (como una cadena), intenta unirse a la red, devolviendo una promesa que se resuelve si tiene éxito, y se rechaza si la autenticación falla. Lo primero que necesitamos es una forma de envolver una promesa para que se rechace automáticamente después de transcurrir demasiado tiempo, de manera que podamos avanzar rápidamente si el punto de acceso no responde.

```
function withTimeout(promise, tiempo) {  
  return new Promise((resolve, reject) => {  
    promise.then(resolve, reject);  
    setTimeout(() => reject("Se agotó el tiempo"), tiempo);  
  });  
}
```

Esto aprovecha el hecho de que una promesa solo puede resolverse o rechazarse una vez: si la promesa dada como argumento se resuelve

o se rechaza primero, ese será el resultado de la promesa devuelta por `withTimeout`. Si, por otro lado, el `setTimeout` se ejecuta primero, rechazando la promesa, se ignoran cualquier llamada posterior a `resolve` o `reject`.

Para encontrar todo el código de acceso, necesitamos buscar repetidamente el siguiente dígito probando cada dígito. Si la autenticación tiene éxito, sabremos que hemos encontrado lo que buscamos. Si falla inmediatamente, sabremos que ese dígito era incorrecto y debemos probar con el siguiente. Si la solicitud se agota, hemos encontrado otro dígito correcto y debemos continuar agregando otro dígito. Debido a que no puedes esperar una promesa dentro de un bucle `for`, Carla utiliza una función recursiva para llevar a cabo este proceso. En cada llamada, obtiene el código tal como lo conocemos hasta ahora, así como el siguiente dígito a probar. Dependiendo de lo que suceda, puede devolver un código terminado, o llamar de nuevo a sí misma, ya sea para comenzar a descifrar la siguiente posición en el código, o para intentarlo de nuevo con otro dígito.

```
function crackPasscode(networkID) {  
  function nextDigit(code, digit) {  
    let newCode = code + digit;  
    return withTimeout(joinWifi(networkID, newCode), 50)  
      .then(() => newCode)  
      .catch(failure => {  
        if (failure == "Timed out") {  
          return nextDigit(newCode, 0);  
        } else if (digit < 9) {  
          return nextDigit(code, digit + 1);  
        } else {  
          throw failure;  
        }  
      })  
  };  
};
```



```
    }  
    return nextDigit("", 0);  
}
```

El punto de acceso suele responder a solicitudes de autenticación incorrectas en aproximadamente 20 milisegundos, por lo que, para estar seguros, esta función espera 50 milisegundos antes de hacer expirar una solicitud.

```
crackPasscode("HANGAR 2").then(console.log);  
// → 555555
```

Carla inclina la cabeza y suspira. Esto habría sido más satisfactorio si el código hubiera sido un poco más difícil de adivinar.

FUNCIONES ASÍNCRONAS

Incluso con promesas, este tipo de código asíncrono es molesto de escribir. Las promesas a menudo necesitan ser encadenadas de manera verbosa y arbitraria. Y nos vimos obligados a introducir una función recursiva solo para crear un bucle.

Lo que la función de descifrado realmente hace es completamente lineal: siempre espera a que la acción anterior se complete antes de comenzar la siguiente. En un modelo de programación síncrona, sería más sencillo de expresar.

La buena noticia es que JavaScript te permite escribir código pseudo-sincrónico para describir la computación asíncrona. Una función `async` es una función que implícitamente devuelve una promesa y que puede, en su cuerpo, `await` otras promesas de una manera que *parece* sincrónica.

Podemos reescribir `crackPasscode` de la siguiente manera:

```
async function crackPasscode(networkID) {
  for (let code = "";;) {
    for (let digit = 0;; digit++) {
      let newCode = code + digit;
      try {
        await withTimeout(joinWifi(networkID, newCode), 50);
        return newCode;
      } catch (failure) {
        if (failure == "Timed out") {
          code = newCode;
          break;
        } else if (digit == 9) {
          throw failure;
        }
      }
    }
  }
}
```

Esta versión muestra de manera más clara la estructura de doble bucle de la función (el bucle interno prueba el dígito 0 al 9, el bucle externo añade dígitos al código de acceso).

Una función `async` está marcada con la palabra `async` antes de la palabra clave `function`. Los métodos también pueden ser marcados como `async` escribiendo `async` antes de su nombre. Cuando se llama a una función o método de esta manera, devuelve una promesa. Tan pronto como la función devuelve algo, esa promesa se resuelve. Si el cuerpo genera una excepción, la promesa es rechazada.

Dentro de una función `async`, la palabra `await` puede colocarse delante de una expresión para esperar a que una promesa se resuelva

y luego continuar con la ejecución de la función. Si la promesa es rechazada, se genera una excepción en el punto del `await`.

Una función así ya no se ejecuta, como una función regular de JavaScript, de principio a fin de una sola vez. En su lugar, puede estar *congelada* en cualquier punto que tenga un `await`, y puede continuar más tarde.

Para la mayoría del código asíncrono, esta notación es más conveniente que usar directamente promesas. Aún necesitas comprender las promesas, ya que en muchos casos todavía interactúas con ellas directamente. Pero al encadenarlas, las funciones `async` suelen ser más agradables de escribir que encadenar llamadas `then`.

GENERADORES

Esta capacidad de pausar y luego reanudar funciones no es exclusiva de las funciones `async`. JavaScript también tiene una característica llamada *generator functions*. Son similares, pero sin las promesas.

Cuando defines una función con `function*` (colocando un asterisco después de la palabra `function`), se convierte en un generador. Al llamar a un generador, devuelve un iterador, que ya vimos en [Capítulo 6](#).

```
function* powers(n) {  
  for (let current = n;; current *= n) {  
    yield current;  
  }  
}  
  
for (let power of powers(3)) {
```

```
    if (power > 50) break;
    console.log(power);
}
// → 3
// → 9
// → 27
```

Inicialmente, al llamar a `powers`, la función se congela desde el principio. Cada vez que llamas a `next` en el iterador, la función se ejecuta hasta que encuentra una expresión `yield`, que la pausa y hace que el valor generado se convierta en el próximo valor producido por el iterador. Cuando la función retorna (la del ejemplo nunca lo hace), el iterador ha terminado.

Escribir iteradores a menudo es mucho más fácil cuando usas funciones generadoras. El iterador para la clase `Group` (del ejercicio en [Capítulo 6](#)) se puede escribir con este generador:

```
Group.prototype[Symbol.iterator] = function*() {
  for (let i = 0; i < this.members.length; i++) {
    yield this.members[i];
  }
};
```

Ya no es necesario crear un objeto para mantener el estado de la iteración: los generadores guardan automáticamente su estado local cada vez que hacen un `yield`.

Tales expresiones `yield` solo pueden ocurrir directamente en la función generadora misma y no en una función interna que definas dentro de ella. El estado que un generador guarda, al hacer `yield`, es solo su entorno *local* y la posición donde hizo el `yield`.

Una función `async` es un tipo especial de generador. Produce una promesa al llamarla, la cual se resuelve cuando retorna (termina) y se rechaza cuando arroja una excepción. Cada vez que hace un `yield` (awaits) una promesa, el resultado de esa promesa (valor o excepción generada) es el resultado de la expresión `await`.

UN PROYECTO DE ARTE DE CORVIDOS

Esta mañana, Carla se despertó con un ruido desconocido en la pista de aterrizaje fuera de su hangar. Saltando al borde del techo, ve que los humanos están preparando algo. Hay muchos cables eléctricos, un escenario y una especie de gran pared negra que están construyendo.

Siendo una cuerva curiosa, Carla echa un vistazo más de cerca a la pared. Parece estar compuesta por varios dispositivos grandes con frente de vidrio conectados a cables. En la parte trasera, los dispositivos dicen “LedTec SIG-5030”.

Una rápida búsqueda en Internet saca a relucir un manual de usuario para estos dispositivos. Parecen ser señales de tráfico, con una matriz programable de luces LED ambarinas. La intención de los humanos probablemente sea mostrar algún tipo de información en ellas durante su evento. Curiosamente, las pantallas pueden ser programadas a través de una red inalámbrica. ¿Podría ser que estén conectadas a la red local del edificio?

Cada dispositivo en una red recibe una *dirección IP*, que otros dispositivos pueden usar para enviarle mensajes. Hablamos más sobre eso en el [Capítulo 13](#). Carla nota que sus propios teléfonos reciben direcciones como `10.0.0.20` o `10.0.0.33`. Podría valer la

pena intentar enviar mensajes a todas esas direcciones y ver si alguna responde a la interfaz descrita en el manual de las señales.

El [Capítulo 18](#) muestra cómo hacer solicitudes reales en redes reales. En este capítulo, usaremos una función ficticia simplificada llamada `request` para la comunicación en red. Esta función toma dos argumentos: una dirección de red y un mensaje, que puede ser cualquier cosa que se pueda enviar como JSON, y devuelve una promesa que se resuelve con una respuesta de la máquina en la dirección dada, o se rechaza si hubo un problema.

Según el manual, puedes cambiar lo que se muestra en una señal SIG-5030 enviándole un mensaje con contenido como `{"command": "display", "data": [0, 0, 3, ...]}`, donde `data` contiene un número por cada punto de LED, indicando su brillo; 0 significa apagado, 3 significa brillo máximo. Cada señal tiene 50 luces de ancho y 30 luces de alto, por lo que un comando de actualización debe enviar 1500 números.

Este código envía un mensaje de actualización de pantalla a todas las direcciones en la red local para ver cuál se queda. Cada uno de los números en una dirección IP puede ir de 0 a 255. En los datos que envía, activa un número de luces correspondiente al último número de la dirección de red.

```
for (let addr = 1; addr < 256; addr++) {  
  let data = [];  
  for (let n = 0; n < 1500; n++) {  
    data.push(n < addr ? 3 : 0);  
  }  
  let ip = `10.0.0.${addr}`;  
  request(ip, {command: "display", data})  
    .then(() => console.log(`Solicitud a ${ip} aceptada`))
```

```
    .catch(() => {});  
}
```

Dado que la mayoría de estas direcciones no existirán o no aceptarán tales mensajes, la llamada a `catch` se asegura de que los errores de red no hagan que el programa falle. Las solicitudes se envían todas inmediatamente, sin esperar a que otras solicitudes terminen, para no perder tiempo cuando algunas de las máquinas no respondan.

Después de haber iniciado su exploración de red, Carla regresa afuera para ver el resultado. Para su deleite, todas las pantallas ahora muestran una franja de luz en sus esquinas superiores izquierdas. Están en la red local y sí aceptan comandos. Rápidamente toma nota de los números mostrados en cada pantalla. Hay 9 pantallas, dispuestas tres en alto y tres en ancho. Tienen las siguientes direcciones de red:

```
const screenAddresses = [  
  "10.0.0.44", "10.0.0.45", "10.0.0.41",  
  "10.0.0.31", "10.0.0.40", "10.0.0.42",  
  "10.0.0.48", "10.0.0.47", "10.0.0.46"  
];
```

Ahora esto abre posibilidades para todo tipo de travesuras. Podría mostrar “los cuervos mandan, los humanos babean” en la pared en letras gigantes. Pero eso se siente un poco grosero. En su lugar, planea mostrar un video de un cuervo volando que cubre todas las pantallas por la noche.

Carla encuentra un clip de video adecuado, en el cual un segundo y medio de metraje se puede repetir para crear un video en bucle mostrando el aleteo de un cuervo. Para ajustarse a las nueve

pantallas (cada una de las cuales puede mostrar 50 por 30 píxeles), Carla corta y redimensiona los videos para obtener una serie de imágenes de 150 por 90, diez por segundo. Estas luego se cortan en nueve rectángulos cada una, y se procesan para que los puntos oscuros en el video (donde está el cuervo) muestren una luz brillante, y los puntos claros (sin cuervo) permanezcan oscuros, lo que debería crear el efecto de un cuervo ámbar volando contra un fondo negro.

Ella ha configurado la variable `clipImages` para contener un array de fotogramas, donde cada fotograma se representa con un array de nueve conjuntos de píxeles, uno para cada pantalla, en el formato que los letreros esperan.

Para mostrar un único fotograma del video, Carla necesita enviar una solicitud a todas las pantallas a la vez. Pero también necesita esperar el resultado de estas solicitudes, tanto para no comenzar a enviar el siguiente fotograma antes de que el actual se haya enviado correctamente, como para notar cuando las solicitudes están fallando.

Promise tiene un método estático `all` que se puede usar para convertir un array de promesas en una sola promesa que se resuelve en un array de resultados. Esto proporciona una forma conveniente de que algunas acciones asíncronas sucedan al lado unas de otras, esperar a que todas terminen y luego hacer algo con sus resultados (o al menos esperar a que terminen para asegurarse de que no fallen).

```
function displayFrame(frame) {  
  return Promise.all(frame.map((data, i) => {  
    return request(screenAddresses[i], {  
      command: "display",  
      data
```



```

    });
  });
}

```

Esto recorre las imágenes en `frame` (que es un array de arrays de datos de visualización) para crear un array de promesas de solicitud. Luego devuelve una promesa que combina todas esas promesas.

Para poder detener un video en reproducción, el proceso está envuelto en una clase. Esta clase tiene un método asíncrono `play` que devuelve una promesa que solo se resuelve cuando la reproducción se detiene de nuevo a través del método `stop`.

```

function wait(time) {
  return new Promise(accept => setTimeout(accept, time));
}

class VideoPlayer {
  constructor(frames, frameTime) {
    this.frames = frames;
    this.frameTime = frameTime;
    this.stopped = true;
  }

  async play() {
    this.stopped = false;
    for (let i = 0; !this.stopped; i++) {
      let nextFrame = wait(this.frameTime);
      await displayFrame(this.frames[i % this.frames.length]);
      await nextFrame;
    }
  }

  stop() {
    this.stopped = true;
  }
}

```

La función `wait` envuelve `setTimeout` en una promesa que se resuelve después del número de milisegundos especificado. Esto es útil para controlar la velocidad de reproducción.

```
let video = new VideoPlayer(clipImages, 100);
video.play().catch(e => {
  console.log("La reproducción falló: " + e);
});
setTimeout(() => video.stop(), 15000);
```

Durante toda la semana que dura el muro de pantalla, todas las noches, cuando está oscuro, aparece misteriosamente un enorme pájaro naranja brillante en él.

EL BUCLE DE EVENTOS

Un programa asincrónico comienza ejecutando su script principal, que a menudo configurará devoluciones de llamada para ser llamadas más tarde. Ese script principal, así como las devoluciones de llamada, se ejecutan por completo de una vez, sin interrupciones. Pero entre ellos, el programa puede estar inactivo, esperando a que ocurra algo.

Por lo tanto, las devoluciones de llamada no son llamadas directamente por el código que las programó. Si llamo a `setTimeout` desde dentro de una función, esa función ya habrá retornado en el momento en que se llame a la función de devolución de llamada. Y cuando la devolución de llamada regresa, el control no vuelve a la función que lo programó.

El comportamiento asincrónico ocurre en su propia función vacía pila de llamadas. Esta es una de las razones por las que, sin

promesas, gestionar excepciones en código asíncrono es tan difícil. Dado que cada devolución de llamada comienza con una pila de llamadas en su mayoría vacía, sus manejadores de `catch` no estarán en la pila cuando lancen una excepción.

```
try {
  setTimeout(() => {
    throw new Error("¡Zoom!");
  }, 20);
} catch (e) {
  // Esto no se ejecutará
  console.log("Atrapado", e);
}
```

No importa cuán cerca ocurran eventos, como tiempos de espera o solicitudes entrantes, un entorno JavaScript ejecutará solo un programa a la vez. Puedes pensar en esto como ejecutar un gran bucle *alrededor* de tu programa, llamado el *bucle de eventos*. Cuando no hay nada que hacer, ese bucle se pausa. Pero a medida que llegan eventos, se agregan a una cola y su código se ejecuta uno tras otro. Debido a que no se ejecutan dos cosas al mismo tiempo, un código lento puede retrasar el manejo de otros eventos.

Este ejemplo establece un tiempo de espera pero luego se demora hasta después del momento previsto para el tiempo de espera, provocando que el tiempo de espera sea tardío.

```
let start = Date.now();
setTimeout(() => {
  console.log("El tiempo de espera se ejecutó en", Date.now() -
start);
}, 20);
while (Date.now() < start + 50) {}
console.log("Tiempo perdido hasta", Date.now() - start);
```

```
// → Tiempo perdido hasta 50
// → El tiempo de espera se ejecutó en 55
```

Las promesas siempre se resuelven o se rechazan como un nuevo evento. Incluso si una promesa ya está resuelta, esperarla hará que su devolución de llamada se ejecute después de que termine el script actual, en lugar de inmediatamente.

```
Promise.resolve("Hecho").then(console.log);
console.log("¡Yo primero!");
// → ¡Yo primero!
// → Hecho
```

En capítulos posteriores veremos varios tipos de eventos que se ejecutan en el bucle de eventos.

ERRORES ASINCRÓNICOS

Cuando tu programa se ejecuta de forma síncrona, de una sola vez, no hay cambios de estado ocurriendo excepto aquellos que el programa mismo realiza. Para programas asíncronos esto es diferente, pueden tener *brechas* en su ejecución durante las cuales otro código puede correr.

Veamos un ejemplo. Esta es una función que intenta reportar el tamaño de cada archivo en un arreglo de archivos, asegurándose de leerlos todos al mismo tiempo en lugar de en secuencia.

```
async function fileSizes(files) {
  let lista = "";
  await Promise.all(files.map(async fileName => {
    lista += fileName + ": " +
      (await textFile(fileName)).length + "\n";
  }));
}
```

```
    return lista;
}
```

La parte `async fileName =>` muestra cómo también se pueden hacer arrow functions `async` colocando la palabra `async` delante de ellas.

El código no parece ser sospechoso de inmediato... mapea la función flecha `async` sobre el arreglo de nombres, creando un arreglo de promesas, y luego usa `Promise.all` para esperar a todas ellas antes de devolver la lista que construyen.

Pero está totalmente roto. Siempre devolverá solo una línea de salida, enumerando el archivo que tardó más en leer.

¿Puedes descubrir por qué?

El problema radica en el operador `+=`, que toma el valor *actual* de `lista` en el momento en que comienza a ejecutarse la instrucción y luego, cuando el `await` termina, establece el enlace `lista` como ese valor más la cadena agregada.

Pero entre el momento en que comienza a ejecutarse la instrucción y el momento en que termina, hay una brecha asincrónica. La expresión `map` se ejecuta antes de que se agregue cualquier cosa a la lista, por lo que cada uno de los operadores `+=` comienza desde una cadena vacía y termina, cuando termina su recuperación de almacenamiento, estableciendo `lista` en el resultado de agregar su línea a la cadena vacía.

Esto podría haberse evitado fácilmente devolviendo las líneas de las promesas mapeadas y llamando a `join` en el resultado de

`Promise.all`, en lugar de construir la lista cambiando un enlace. Como suele ser, calcular nuevos valores es menos propenso a errores que cambiar valores existentes.

```
async function fileSizes(files) {
  let líneas = files.map(async fileName => {
    return fileName + ": " +
      (await textFile(fileName)).length;
  });
  return (await Promise.all(líneas)).join("\n");
}
```

Errores como este son fáciles de cometer, especialmente al usar `await`, y debes ser consciente de dónde ocurren las brechas en tu código. Una ventaja de la asincronía *explícita* de JavaScript (ya sea a través de devoluciones de llamada, promesas o `await`) es que identificar estas brechas es relativamente fácil.

RESUMEN

La programación asincrónica hace posible expresar la espera de acciones de larga duración sin congelar todo el programa. Los entornos de JavaScript típicamente implementan este estilo de programación utilizando devoluciones de llamada, funciones que se llaman cuando las acciones se completan. Un bucle de eventos programa estas devoluciones de llamada para que se llamen cuando sea apropiado, una tras otra, de modo que su ejecución no se superponga. La programación de forma asíncrona se facilita gracias a las promesas, que son objetos que representan acciones que podrían completarse en el futuro, y las funciones `async`, que te permiten escribir un programa asíncrono como si fuera sincrónico.

EJERCICIOS

MOMENTOS DE TRANQUILIDAD

Hay una cámara de seguridad cerca del laboratorio de Carla que se activa con un sensor de movimiento. Está conectada a la red y comienza a enviar un flujo de video cuando está activa. Como prefiere no ser descubierta, Carla ha configurado un sistema que detecta este tipo de tráfico de red inalámbrico y enciende una luz en su guarida cada vez que hay actividad afuera, para que ella sepa cuándo mantenerse en silencio.

También ha estado registrando los momentos en que la cámara se activa desde hace un tiempo, y quiere utilizar esta información para visualizar qué momentos, en una semana promedio, tienden a ser tranquilos y cuáles tienden a ser ocupados. El registro se almacena en archivos que contienen un número de marca de tiempo por línea (como devuelto por `Date.now()`).

```
1695709940692
1695701068331
1695701189163
```

El archivo `"camera_logs.txt"` contiene una lista de archivos de registro. Escribe una función asíncrona `activityTable(día)` que, para un día de la semana dado, devuelva un array de 24 números, uno para cada hora del día, que contenga la cantidad de observaciones de tráfico de red de la cámara vista en esa hora del día. Los días se identifican por número utilizando el sistema utilizado por `Date.getDay`, donde el domingo es 0 y el sábado es 6.

La función `activityGraph`, proporcionada por el `sandbox`, resume dicha tabla en una cadena.

Utiliza la función `textFile` definida anteriormente, que al recibir un nombre de archivo devuelve una promesa que se resuelve en el contenido del archivo. Recuerda que `new Date(marcaDeTiempo)` crea un objeto `Date` para ese momento, que tiene métodos `getDay` y `getHours` que devuelven el día de la semana y la hora del día.

Ambos tipos de archivos, la lista de archivos de registro y los propios archivos de registro, tienen cada dato en su propia línea, separados por caracteres de nueva línea (`"\n"`).

CONSTRUYENDO `PROMISE.ALL`

Como vimos, dado un array de promesas, `Promise.all` devuelve una promesa que espera a que todas las promesas en el array finalicen. Luego tiene éxito, devolviendo un array de valores de resultado. Si una promesa en el array falla, la promesa devuelta por `all` también falla, con la razón de fallo de la promesa que falló.

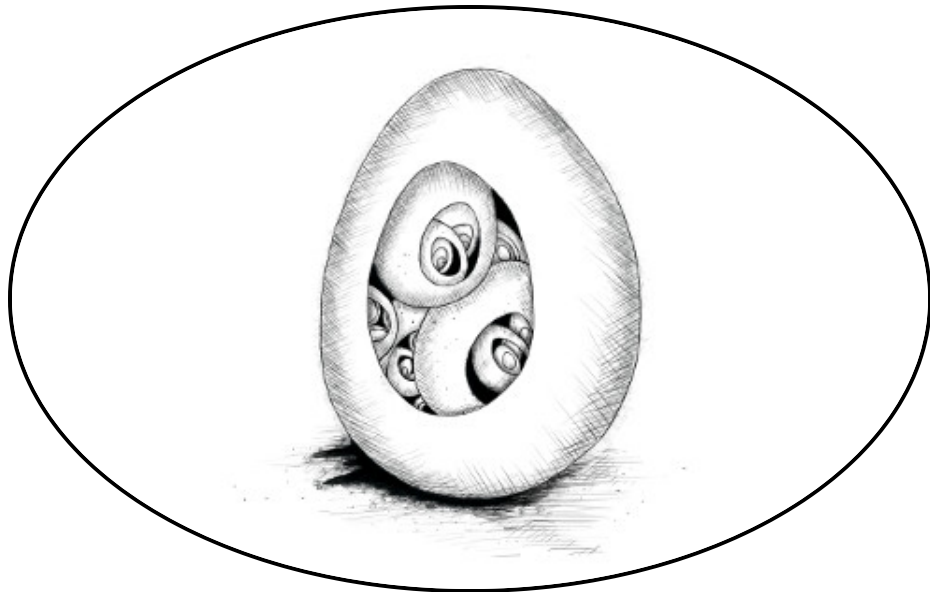
Implementa algo similar tú mismo como una función regular llamada `Promise_all`.

Recuerda que después de que una promesa tiene éxito o falla, no puede volver a tener éxito o fallar, y las llamadas posteriores a las funciones que la resuelven se ignoran. Esto puede simplificar la forma en que manejas el fallo de tu promesa.

PROYECTO: UN LENGUAJE DE PROGRAMACIÓN

“El evaluador, que determina el significado de expresiones en un lenguaje de programación, es solo otro programa.”

—Hal Abelson y Gerald Sussman, *Estructura e Interpretación de Programas de Computadora*



Crear tu propio lenguaje de programación es sorprendentemente fácil (si no apuntas muy alto) y muy esclarecedor.

Lo principal que quiero mostrar en este capítulo es que no hay magia involucrada en la construcción de un lenguaje de programación. A menudo he sentido que algunas invenciones humanas eran tan inmensamente inteligentes y complicadas que nunca las entendería.

Pero con un poco de lectura y experimentación, a menudo resultan ser bastante mundanas.

Construiremos un lenguaje de programación llamado Egg. Será un lenguaje simple y diminuto, pero lo suficientemente poderoso como para expresar cualquier cálculo que puedas imaginar. Permitirá una simple abstracción basada en funciones.

ANÁLISIS SINTÁCTICO

La parte más inmediatamente visible de un lenguaje de programación es su *sintaxis*, o notación. Un *analizador sintáctico* es un programa que lee un fragmento de texto y produce una estructura de datos que refleja la estructura del programa contenido en ese texto. Si el texto no forma un programa válido, el analizador sintáctico debería señalar el error.

Nuestro lenguaje tendrá una sintaxis simple y uniforme. Todo en Egg es una expresión. Una expresión puede ser el nombre de una asignación, un número, una cadena o una *aplicación*. Las aplicaciones se utilizan para llamadas de funciones pero también para estructuras como `if` o `while`.

Para mantener el analizador sintáctico simple, las cadenas en Egg no admiten nada parecido a los escapes con barra invertida. Una cadena es simplemente una secuencia de caracteres que no son comillas dobles, envueltos entre comillas dobles. Un número es una secuencia de dígitos. Los nombres de las asignaciones pueden consistir en cualquier carácter que no sea espacio en blanco y que no tenga un significado especial en la sintaxis.

Las aplicaciones se escriben de la misma manera que en JavaScript, colocando paréntesis después de una expresión y teniendo cualquier número de argumentos entre esos paréntesis, separados por comas.

```
do(define(x, 10),  
    if(>(x, 5),  
        print("grande"),  
        print("pequeño")))
```

La uniformidad del lenguaje Egg significa que las cosas que son operadores en JavaScript (como `>`) son asignaciones normales en este lenguaje, aplicadas de la misma manera que otras funciones. Y dado que la sintaxis no tiene concepto de bloque, necesitamos un constructo `do` para representar la realización de múltiples tareas en secuencia.

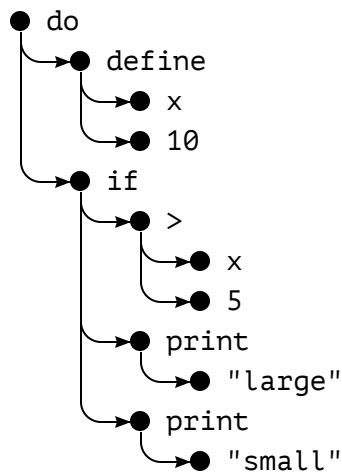
La estructura de datos que el analizador sintáctico utilizará para describir un programa consiste en objetos expresión, cada uno de los cuales tiene una propiedad `type` que indica el tipo de expresión que es y otras propiedades para describir su contenido.

Las expresiones de tipo `"value"` representan cadenas literales o números. Su propiedad `value` contiene el valor de cadena o número que representan. Las expresiones de tipo `"word"` se utilizan para identificadores (nombres). Estos objetos tienen una propiedad `name` que contiene el nombre del identificador como cadena. Finalmente, las expresiones `"apply"` representan aplicaciones. Tienen una propiedad `operator` que se refiere a la expresión que se está aplicando, así como una propiedad `args` que contiene una serie de expresiones de argumento.

La parte `>(x, 5)` del programa anterior se representaría de la siguiente manera:

```
{
  type: "apply",
  operator: {type: "word", name: ">"},
  args: [
    {type: "word", name: "x"},
    {type: "value", value: 5}
  ]
}
```

Esta estructura de datos se llama un *árbol de sintaxis*. Si te imaginas los objetos como puntos y los enlaces entre ellos como líneas entre esos puntos, tiene una forma similar a un árbol. El hecho de que las expresiones contienen otras expresiones, que a su vez pueden contener más expresiones, es similar a la forma en que las ramas de un árbol se dividen y vuelven a dividir.



Contrasta esto con el analizador que escribimos para el formato de archivo de configuración en [Capítulo 9](#), que tenía una estructura simple: dividía la entrada en líneas y manejaba esas líneas una a la vez. Solo había algunas formas simples que una línea podía tener.

Aquí debemos encontrar un enfoque diferente. Las expresiones no están separadas en líneas, y tienen una estructura recursiva. Las expresiones de aplicación *contienen* otras expresiones.

Afortunadamente, este problema puede resolverse muy bien escribiendo una función de análisis sintáctico que sea recursiva de una manera que refleje la naturaleza recursiva del lenguaje.

Definimos una función `parseExpression`, que recibe una cadena como entrada y devuelve un objeto que contiene la estructura de datos de la expresión al inicio de la cadena, junto con la parte de la cadena que queda después de analizar esta expresión. Al analizar subexpresiones (el argumento de una aplicación, por ejemplo), esta función puede ser llamada nuevamente, obteniendo la expresión de argumento así como el texto que queda. Este texto a su vez puede contener más argumentos o puede ser el paréntesis de cierre que finaliza la lista de argumentos. Esta es la primera parte del analizador sintáctico:

```
function parseExpression(program) {
  program = skipSpace(program);
  let match, expr;
  if (match = /^"([^"]*)"/.exec(program)) {
    expr = {type: "value", value: match[1]};
  } else if (match = /^(\d+(\b)/.exec(program)) {
    expr = {type: "value", value: Number(match[0])};
  } else if (match = /^[^\s(),#"]+/.exec(program)) {
    expr = {type: "word", name: match[0]};
  } else {
    throw new SyntaxError("Sintaxis inesperada: " + program);
  }

  return parseApply(expr, program.slice(match[0].length));
}
```

```
function skipSpace(string) {
  let first = string.search(/\S/);
  if (first == -1) return "";
  return string.slice(first);
}
```

Debido a que Egg, al igual que JavaScript, permite cualquier cantidad de espacios en blanco entre sus elementos, debemos cortar repetidamente el espacio en blanco del inicio de la cadena del programa. Eso es para lo que sirve la función `skipSpace`.

Después de omitir cualquier espacio inicial, `parseExpression` utiliza tres expresiones regulares para detectar los tres elementos atómicos que admite Egg: cadenas, números y palabras. El analizador construye un tipo diferente de estructura de datos dependiendo de cuál de ellos coincida. Si la entrada no coincide con ninguna de estas tres formas, no es una expresión válida y el analizador genera un error. Utilizamos el constructor `SyntaxError` aquí. Esta es una clase de excepción definida por el estándar, al igual que `Error`, pero más específica.

Luego cortamos la parte que coincidió de la cadena del programa y la pasamos, junto con el objeto de la expresión, a `parseApply`, que verifica si la expresión es una aplicación. Si lo es, analiza una lista de argumentos entre paréntesis.

```
function parseApply(expr, program) {
  program = skipSpace(program);
  if (program[0] != "(") {
    return {expr: expr, rest: program};
  }

  program = skipSpace(program.slice(1));
  expr = {type: "apply", operator: expr, args: []};
```

```

while (program[0] != ")") {
  let arg = parseExpression(program);
  expr.args.push(arg.expr);
  program = skipSpace(arg.rest);
  if (program[0] == ",") {
    program = skipSpace(program.slice(1));
  } else if (program[0] != ")") {
    throw new SyntaxError("Se esperaba ',' o ')'");
  }
}
return parseApply(expr, program.slice(1));
}

```

Si el próximo carácter en el programa no es un paréntesis de apertura, esto no es una aplicación y `parseApply` devuelve la expresión que se le dio.

De lo contrario, se salta el paréntesis de apertura y crea el objeto árbol sintáctico para esta expresión de aplicación. Luego llama recursivamente a `parseExpression` para analizar cada argumento hasta encontrar un paréntesis de cierre. La recursión es indirecta, a través de `parseApply` y `parseExpression` llamándose mutuamente.

Dado que una expresión de aplicación puede a su vez ser aplicada (como en `multiplicador(2)(1)`), `parseApply` debe, después de analizar una aplicación, llamarse a sí misma nuevamente para verificar si sigue otro par de paréntesis.

Esto es todo lo que necesitamos para analizar Egg. Lo envolvemos en una conveniente `parse` función que verifica que ha llegado al final de la cadena de entrada después de analizar la expresión (un programa Egg es una sola expresión), y que nos da la estructura de datos del programa.


```

function parse(program) {
  let {expr, rest} = parseExpression(program);
  if (skipSpace(rest).length > 0) {
    throw new SyntaxError("Texto inesperado después del
programa");
  }
  return expr;
}

console.log(parse("(a, 10)"));
// → {type: "apply",
//     operator: {type: "word", name: "+"},
//     args: [{type: "word", name: "a"},
//            {type: "value", value: 10}]}

```

¡Funciona! No nos da información muy útil cuando falla y no almacena la línea y la columna en las que comienza cada expresión, lo cual podría ser útil al informar errores más tarde, pero es suficiente para nuestros propósitos.

EL EVALUADOR

¿Qué podemos hacer con el árbol de sintaxis de un programa?

¡Ejecutarlo, por supuesto! Y eso es lo que hace el evaluador. Le das un árbol de sintaxis y un objeto de ámbito que asocia nombres con valores, y evaluará la expresión que representa el árbol y devolverá el valor que esto produce.

```

const specialForms = Object.create(null);

function evaluate(expr, scope) {
  if (expr.type == "value") {
    return expr.value;
  } else if (expr.type == "word") {
    if (expr.name in scope) {
      return scope[expr.name];
    } else {

```

```

        throw new ReferenceError(
            `Vinculación indefinida: ${expr.name}`);
    }
} else if (expr.type == "apply") {
    let {operator, args} = expr;
    if (operator.type == "word" &&
        operator.name in specialForms) {
        return specialForms[operator.name](expr.args, scope);
    } else {
        let op = evaluate(operator, scope);
        if (typeof op == "function") {
            return op(...args.map(arg => evaluate(arg, scope)));
        } else {
            throw new TypeError("Aplicando una no-función.");
        }
    }
}
}
}

```

El evaluador tiene código para cada uno de los tipos de expresión. Una expresión de valor literal produce su valor. (Por ejemplo, la expresión `100` simplemente se evalúa como el número 100.) Para un enlace, debemos verificar si está realmente definido en el ámbito y, si lo está, obtener el valor del enlace.

Las aplicaciones son más complicadas. Si son una forma especial, como `if`, no evaluamos nada y pasamos las expresiones de argumento, junto con el ámbito, a la función que maneja esta forma. Si es una llamada normal, evaluamos el operador, verificamos que sea una función, y la llamamos con los argumentos evaluados.

Usamos valores de función JavaScript simples para representar los valores de función de Egg. Volveremos a esto [más tarde](#), cuando se defina la forma especial llamada `fun`.

La estructura recursiva de `evaluate` se asemeja a la estructura similar del analizador sintáctico, y ambos reflejan la estructura del lenguaje en sí. También sería posible combinar el analizador sintáctico y el evaluador en una sola función, y evaluar durante el análisis sintáctico. Pero dividirlos de esta manera hace que el programa sea más claro y flexible.

Esto es realmente todo lo que se necesita para interpretar Egg. Es así de simple. Pero sin definir algunas formas especiales y agregar algunos valores útiles al entorno, todavía no puedes hacer mucho con este lenguaje.

FORMAS ESPECIALES

El objeto `specialForms` se utiliza para definir sintaxis especial en Egg. Asocia palabras con funciones que evalúan dichas formas. Actualmente está vacío. Añadamos `if`.

```
specialForms.if = (args, scope) => {  
  if (args.length !== 3) {  
    throw new SyntaxError("Número incorrecto de argumentos para  
if");  
  } else if (evaluate(args[0], scope) !== false) {  
    return evaluate(args[1], scope);  
  } else {  
    return evaluate(args[2], scope);  
  }  
};
```

La construcción `if` de Egg espera exactamente tres argumentos. Evaluará el primero, y si el resultado no es el valor `false`, evaluará el segundo. De lo contrario, se evaluará el tercero. Esta forma `if` se asemeja más al operador ternario `?:` de JavaScript que al `if` de

JavaScript. Es una expresión, no una declaración, y produce un valor, concretamente, el resultado del segundo o tercer argumento.

Egg también difiere de JavaScript en cómo maneja el valor de condición para `if`. No tratará cosas como cero o la cadena vacía como falso, solo el valor preciso `false`.

La razón por la que necesitamos representar `if` como una forma especial, en lugar de una función regular, es que todos los argumentos de las funciones se evalúan antes de llamar a la función, mientras que `if` debe evaluar solo *uno* de sus segundos o terceros argumentos, dependiendo del valor del primero.

La forma `while` es similar.

```
specialForms.while = (args, scope) => {
  if (args.length !== 2) {
    throw new SyntaxError("Número incorrecto de argumentos para
while");
  }
  while (evaluate(args[0], scope) !== false) {
    evaluate(args[1], scope);
  }

  // Dado que undefined no existe en Egg, devolvemos false,
  // por falta de un resultado significativo.
  return false;
};
```

Otro bloque básico es `do`, que ejecuta todos sus argumentos de arriba abajo. Su valor es el valor producido por el último argumento.

```
specialForms.do = (args, scope) => {
  let valor = false;
  for (let arg of args) {
    valor = evaluate(arg, scope);
  }
```

```
    }  
    return valor;  
};
```

Para poder crear vinculaciones y darles nuevos valores, también creamos una forma llamada `define`. Espera una palabra como su primer argumento y una expresión que produzca el valor a asignar a esa palabra como su segundo argumento. Dado que `define`, al igual que todo, es una expresión, debe devolver un valor. Haremos que devuelva el valor que se asignó (como el operador `=` de JavaScript).

```
specialForms.define = (args, scope) => {  
  if (args.length !== 2 || args[0].type !== "word") {  
    throw new SyntaxError("Uso incorrecto de define");  
  }  
  let value = evaluate(args[1], scope);  
  scope[args[0].name] = value;  
  return value;  
};
```

EL ENTORNO

El `scope` aceptado por `evaluate` es un objeto con propiedades cuyos nombres corresponden a los nombres de los bindings y cuyos valores corresponden a los valores a los que esos bindings están ligados. Definamos un objeto para representar el `scope` global.

Para poder usar la construcción `if` que acabamos de definir, necesitamos tener acceso a valores Booleanos. Dado que solo hay dos valores Booleanos, no necesitamos una sintaxis especial para ellos. Simplemente asignamos dos nombres a los valores `true` y `false` y los usamos.

```
const topScope = Object.create(null);
```

```
topScope.true = true;  
topScope.false = false;
```

Ahora podemos evaluar una expresión simple que niega un valor Booleano.

```
let prog = parse(`if(true, false, true)`);  
console.log(evaluate(prog, topScope));  
// → false
```

Para suministrar operadores básicos de aritmética y comparación, también agregaremos algunas funciones al scope. En aras de mantener el código corto, usaremos `Function` para sintetizar un conjunto de funciones de operadores en un bucle, en lugar de definirlas individualmente.

```
for (let op of ["+", "-", "*", "/", "==", "<", ">"]) {  
  topScope[op] = Function("a, b", `return a ${op} b;`);  
}
```

También es útil tener una forma de imprimir valores, por lo que envolveremos `console.log` en una función y la llamaremos `print`.

```
topScope.print = value => {  
  console.log(value);  
  return value;  
};
```

Esto nos proporciona suficientes herramientas elementales para escribir programas simples. La siguiente función proporciona una

forma conveniente de analizar un programa y ejecutarlo en un nuevo ámbito:

```
function run(program) {  
  return evaluate(parse(program), Object.create(topScope));  
}
```

Utilizaremos las cadenas de prototipos de objetos para representar ámbitos anidados para que el programa pueda agregar bindings a su ámbito local sin modificar el ámbito de nivel superior.

```
run(`  
do(define(total, 0),  
  define(count, 1),  
  while(<(count, 11),  
    do(define(total, +(total, count)),  
      define(count, +(count, 1))))),  
  print(total))  
`);  
// → 55
```

Este es el programa que hemos visto varias veces antes, que calcula la suma de los números del 1 al 10, expresado en Egg. Es claramente más feo que el equivalente programa en JavaScript, pero no está mal para un lenguaje implementado en menos de 150 líneas de código.

FUNCIONES

Un lenguaje de programación sin funciones es un pobre lenguaje de programación.

Afortunadamente, no es difícil agregar una construcción `fun`, que trata su último argumento como el cuerpo de la función y utiliza

todos los argumentos anteriores como los nombres de los parámetros de la función.

```
specialForms.fun = (args, scope) => {
  if (!args.length) {
    throw new SyntaxError("Las funciones necesitan un cuerpo");
  }
  let body = args[args.length - 1];
  let params = args.slice(0, args.length - 1).map(expr => {
    if (expr.type !== "word") {
      throw new SyntaxError("Los nombres de los parámetros deben ser palabras");
    }
    return expr.name;
  });

  return function(...args) {
    if (args.length !== params.length) {
      throw new TypeError("Número incorrecto de argumentos");
    }
    let localScope = Object.create(scope);
    for (let i = 0; i < args.length; i++) {
      localScope[params[i]] = args[i];
    }
    return evaluate(body, localScope);
  };
};
```

Las funciones en Egg tienen su propio ámbito local. La función producida por la forma `fun` crea este ámbito local y añade los enlaces de los argumentos a él. Luego evalúa el cuerpo de la función en este ámbito y devuelve el resultado.

```
run(`
do(define(plusOne, fun(a, +(a, 1))),
  print(plusOne(10)))
`);
// → 11
```



```
run(`
do(define(pow, fun(base, exp,
    if(==(exp, 0),
        1,
        *(base, pow(base, -(exp, 1)))))),
    print(pow(2, 10)))
`);
// → 1024
```

COMPILACIÓN

Lo que hemos construido es un intérprete. Durante la evaluación, actúa directamente sobre la representación del programa producido por el analizador sintáctico.

La compilación es el proceso de agregar otro paso entre el análisis sintáctico y la ejecución de un programa, que transforma el programa en algo que puede ser evaluado de manera más eficiente al hacer la mayor cantidad de trabajo posible por adelantado. Por ejemplo, en lenguajes bien diseñados, es obvio, para cada uso de un enlace, a qué enlace se hace referencia, sin ejecutar realmente el programa. Esto se puede utilizar para evitar buscar el enlace por nombre cada vez que se accede, en su lugar, recuperándolo directamente desde una ubicación de memoria predeterminada.

Tradicionalmente, compilar implica convertir el programa a código máquina, el formato en bruto que un procesador de computadora puede ejecutar. Pero cualquier proceso que convierta un programa a una representación diferente se puede considerar como compilación.

Sería posible escribir una estrategia de evaluación alternativa para Egg, una que primero convierte el programa a un programa JavaScript, usa `Function` para invocar el compilador de JavaScript

en él, y luego ejecuta el resultado. Cuando se hace correctamente, esto haría que Egg se ejecutara muy rápido y aún así fuera bastante simple de implementar.

Si te interesa este tema y estás dispuesto a dedicar tiempo a ello, te animo a intentar implementar ese compilador como ejercicio.

HACIENDO TRAMPA

Cuando definimos `if` y `while`, probablemente notaste que eran envoltorios más o menos triviales alrededor del propio `if` y `while` de JavaScript. De manera similar, los valores en Egg son simplemente valores regulares de JavaScript. Cerrar la brecha hacia un sistema más primitivo, como el código máquina que entiende el procesador, requiere más esfuerzo, pero la forma en que funciona se asemeja a lo que estamos haciendo aquí. Aunque el lenguaje de juguete de este capítulo no hace nada que no se pudiera hacer mejor en JavaScript, sí hay situaciones donde escribir pequeños lenguajes ayuda a realizar trabajos reales.

Tal lenguaje no tiene por qué parecerse a un lenguaje de programación típico. Si JavaScript no viniera equipado con expresiones regulares, por ejemplo, podrías escribir tu propio analizador sintáctico y evaluador para expresiones regulares.

O imagina que estás construyendo un programa que permite crear rápidamente analizadores sintácticos al proporcionar una descripción lógica del lenguaje que necesitan analizar. Podrías definir una notación específica para eso y un compilador que la convierta en un programa analizador.

```
expr = número | cadena | nombre | aplicación  
  
number = dígito+  
  
name = letra+  
  
string = ''' (! ''')* '''  
  
application = expr '(' (expr (',' expr)*)? ')'
```

Esto es lo que comúnmente se denomina un *lenguaje específico de dominio*, un lenguaje diseñado para expresar un ámbito estrecho de conocimiento. Tal lenguaje puede ser más expresivo que un lenguaje de propósito general porque está diseñado para describir exactamente las cosas que necesitan ser descritas en su dominio, y nada más.

EJERCICIOS

ARRAYS

Agrega soporte para arrays en Egg añadiendo las siguientes tres funciones al ámbito superior: `array(...valores)` para construir un array que contenga los valores de los argumentos, `length(array)` para obtener la longitud de un array y `element(array, n)` para obtener el n-ésimo elemento de un array.

CLAUSURA

La forma en que hemos definido `fun` permite que las funciones en Egg hagan referencia al ámbito circundante, lo que permite que el cuerpo de la función use valores locales que eran visibles en el

momento en que se definió la función, al igual que lo hacen las funciones de JavaScript.

El siguiente programa ilustra esto: la función `f` devuelve una función que suma su argumento al argumento de `f`, lo que significa que necesita acceder al ámbito local dentro de `f` para poder usar la vinculación `a`.

```
run(`
do(define(f, fun(a, fun(b, +(a, b)))),
  print(f(4)(5)))
`);
// → 9
```

Vuelve a la definición del formulario `fun` y explica qué mecanismo hace que esto funcione.

COMENTARIOS

Sería bueno si pudiéramos escribir comentarios en Egg. Por ejemplo, siempre que encontremos un signo de almohadilla (`#`), podríamos tratar el resto de la línea como un comentario y ignorarlo, similar a `//` en JavaScript.

No tenemos que hacer grandes cambios en el analizador para admitir esto. Simplemente podemos cambiar `skipSpace` para omitir comentarios como si fueran espacios en blanco de manera que todos los puntos donde se llama a `skipSpace` ahora también omitirán comentarios. Realiza este cambio.

CORRIGIENDO EL ÁMBITO

Actualmente, la única forma de asignar un enlace un valor es `define`. Esta construcción actúa como una forma tanto de definir nuevos enlaces como de dar un nuevo valor a los existentes.

Esta ambigüedad causa un problema. Cuando intentas darle un nuevo valor a un enlace no local, terminarás definiendo uno local con el mismo nombre en su lugar. Algunos lenguajes funcionan de esta manera por diseño, pero siempre he encontrado que es una forma incómoda de manejar el ámbito.

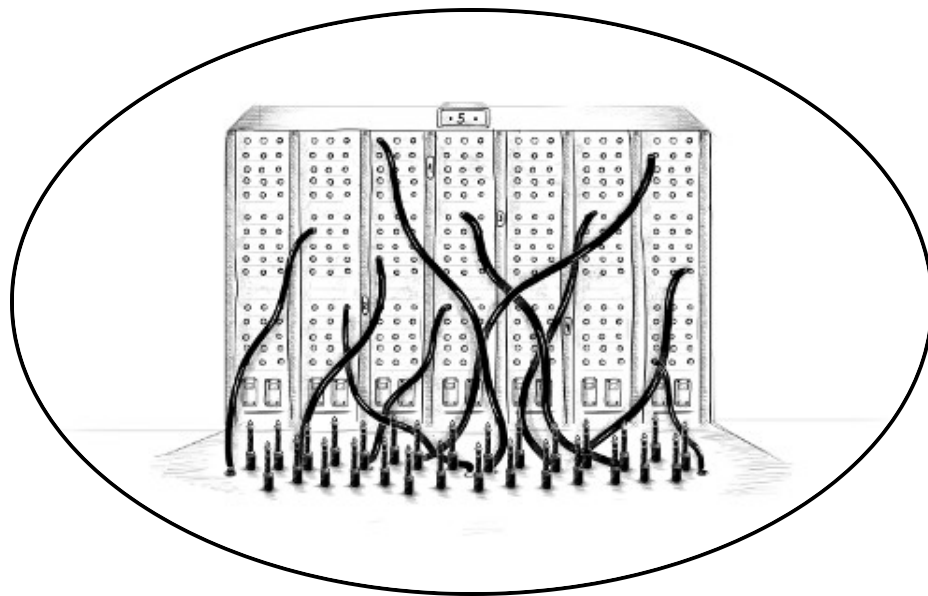
Agrega una forma especial `set`, similar a `define`, que da un nuevo valor a un enlace, actualizando el enlace en un ámbito exterior si aún no existe en el ámbito interior. Si el enlace no está definido en absoluto, lanza un `ReferenceError` (otro tipo de error estándar).

La técnica de representar los ámbitos como objetos simples, que hasta ahora ha sido conveniente, te causará un pequeño problema en este punto. Es posible que desees usar la función `Object.getPrototypeOf`, la cual devuelve el prototipo de un objeto. También recuerda que puedes utilizar `Object.hasOwnProperty` para verificar si un objeto dado tiene una propiedad.

JAVASCRIPT Y EL NAVEGADOR

“El sueño detrás de la Web es de un espacio de información común en el que nos comunicamos compartiendo información. Su universalidad es esencial: el hecho de que un enlace de hipertexto pueda apuntar a cualquier cosa, ya sea personal, local o global, ya sea un borrador o altamente pulido.”

—Tim Berners-Lee, *La World Wide Web: Una historia personal muy breve*



Los próximos capítulos de este libro hablarán sobre los navegadores web. Sin los navegadores web, no habría JavaScript. O incluso si existiera, nadie le habría prestado atención.

La tecnología web ha sido descentralizada desde el principio, no solo técnicamente, sino también en la forma en que evolucionó. Varios fabricantes de navegadores han añadido nueva funcionalidad de manera ad hoc y a veces sin mucho sentido, que luego, a veces,

terminaba siendo adoptada por otros, y finalmente establecida como en los estándares.

Esto es a la vez una bendición y una maldición. Por un lado, es empoderador no tener a una parte central controlando un sistema, sino mejorando con la contribución de diferentes partes que trabajan en una colaboración laxa (o a veces en abierta hostilidad). Por otro lado, la forma caótica en que se desarrolló la Web significa que el sistema resultante no es precisamente un ejemplo brillante de coherencia interna. Algunas partes son directamente confusas y están mal diseñadas.

REDES Y EL INTERNET

Las redes de computadoras existen desde la década de 1950. Si conectas cables entre dos o más computadoras y les permites enviar datos de ida y vuelta a través de estos cables, puedes hacer todo tipo de cosas maravillosas.

Y si conectar dos máquinas en el mismo edificio nos permite hacer cosas maravillosas, conectar máquinas en todo el planeta debería ser aún mejor. La tecnología para comenzar a implementar esta visión se desarrolló en la década de 1980, y la red resultante se llama el *Internet*. Ha cumplido su promesa.

Una computadora puede usar esta red para enviar bits a otra computadora. Para que surja una comunicación efectiva de este envío de bits, las computadoras en ambos extremos deben saber qué se supone que representan los bits. El significado de cualquier secuencia dada de bits depende enteramente del tipo de cosa que está tratando de expresar y del mecanismo de codificación utilizado.

Un *protocolo de red* describe un estilo de comunicación sobre una red. Hay protocolos para enviar correos electrónicos, para recibir correos electrónicos, para compartir archivos e incluso para controlar computadoras que han sido infectadas por software malicioso.

El *Protocolo de Transferencia de Hipertexto* (HTTP) es un protocolo para recuperar recursos nombrados (trozos de información, como páginas web o imágenes). Especifica que el lado que realiza la solicitud debe comenzar con una línea como esta, nombrando el recurso y la versión del protocolo que está intentando usar:

```
GET /index.html HTTP/1.1
```

Hay muchas más reglas sobre la forma en que el solicitante puede incluir más información en la solicitud y la forma en que el otro lado, que devuelve el recurso, empaqueta su contenido. Veremos HTTP con un poco más de detalle en el [Capítulo 18](#).

La mayoría de los protocolos se construyen sobre otros protocolos. HTTP trata la red como un dispositivo similar a un flujo en el que puedes poner bits y hacer que lleguen al destino correcto en el orden correcto. Proporcionar esas garantías encima del envío de datos primitivos que proporciona la red es un problema bastante complicado.

El *Protocolo de Control de Transmisión* (TCP) es un protocolo que aborda este problema. Todos los dispositivos conectados a Internet lo “hablan” y la mayoría de las comunicaciones en Internet se construyen sobre él.

Una conexión TCP funciona de la siguiente manera: una computadora debe estar esperando, o *escuchando*, a que otras computadoras comiencen a hablar con ella. Para poder escuchar diferentes tipos de comunicación al mismo tiempo en una sola máquina, cada oyente tiene asociado un número (llamado *puerto*). La mayoría de los protocolos especifican qué puerto debe usarse de forma predeterminada. Por ejemplo, cuando queremos enviar un correo electrónico usando el protocolo SMTP, se espera que la máquina a través de la cual lo enviamos esté escuchando en el puerto 25.

Otra computadora puede establecer entonces una conexión conectándose a la máquina de destino usando el número de puerto correcto. Si la máquina de destino es alcanzable y está escuchando en ese puerto, la conexión se crea con éxito. La computadora que escucha se llama el *servidor*, y la computadora que se conecta se llama el *cliente*.

Dicha conexión actúa como un conducto bidireccional a través del cual pueden fluir los bits: las máquinas en ambos extremos pueden insertar datos en él. Una vez que los bits se transmiten con éxito, pueden volver a ser leídos por la máquina del otro lado. Este es un modelo conveniente. Se podría decir que TCP proporciona una abstracción de la red.

LA WEB

El *World Wide Web* (no se debe confundir con el Internet en su totalidad) es un conjunto de protocolos y formatos que nos permiten visitar páginas web en un navegador. La parte “Web” en el nombre se

refiere al hecho de que estas páginas pueden enlazarse fácilmente entre sí, conectándose así en una gran malla por la que los usuarios pueden moverse.

Para formar parte de la Web, todo lo que necesitas hacer es conectar una máquina al Internet y hacer que escuche en el puerto 80 con el protocolo HTTP para que otras computadoras puedan solicitarle documentos.

Cada documento en la Web está nombrado por un *Localizador de Recursos Uniforme* (URL), que se ve algo así:

http://eloquentjavascript.net/13_browser.html			
protocol	servidor	ruta	

La primera parte nos dice que esta URL utiliza el protocolo HTTP (en contraposición, por ejemplo, a HTTP cifrado, que sería *https://*). Luego viene la parte que identifica desde qué servidor estamos solicitando el documento. Por último está una cadena de ruta que identifica el documento específico (o *recurso*) en el que estamos interesados.

Las máquinas conectadas a Internet tienen una *dirección IP*, que es un número que se puede utilizar para enviar mensajes a esa máquina, y se ve algo así como 149.210.142.219 o 2001:4860:4860::8888. Pero las listas de números más o menos aleatorios son difíciles de recordar y complicados de escribir, así que en su lugar puedes registrar un *nombre de dominio* para una dirección específica o un conjunto de direcciones. Registré *eloquentjavascript.net* para apuntar a la dirección IP de una

máquina que controlo y, por lo tanto, puedo usar ese nombre de dominio para servir páginas web.

Si escribes esta URL en la barra de direcciones de tu navegador, el navegador intentará recuperar y mostrar el documento en esa URL. Primero, tu navegador tiene que averiguar a qué dirección se refiere *eloquentjavascript.net*. Luego, utilizando el protocolo HTTP, hará una conexión con el servidor en esa dirección y solicitará el recurso */13_browser.html*. Si todo va bien, el servidor enviará un documento, que tu navegador mostrará en tu pantalla.

HTML

HTML, que significa *Lenguaje de Marcado de Hipertexto*, es el formato de documento utilizado para páginas web. Un documento HTML contiene texto, así como *etiquetas* que estructuran el texto, describiendo cosas como enlaces, párrafos y encabezados.

Un documento HTML corto podría lucir así:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Mi página de inicio</title>
  </head>
  <body>
    <h1>Mi página de inicio</h1>
    <p>Hola, soy Marijn y esta es mi página de inicio.</p>
    <p>¡También escribí un libro! Léelo
      <a href="http://eloquentjavascript.net">aquí</a>.</p>
  </body>
</html>
```

Así es como se vería dicho documento en el navegador:

My home page

Hello, I am Marijn and this is my home page.

I also wrote a book! Read it [here](#).

Las etiquetas, encerradas en corchetes angulares (< y >, los símbolos de *menor que* y *mayor que*), proporcionan información sobre la estructura del documento. El otro texto es simplemente texto plano.

El documento comienza con <!doctype html>, lo que indica al navegador interpretar la página como HTML *moderno*, en contraposición a estilos obsoletos que se utilizaban en el pasado.

Los documentos HTML tienen una cabecera y un cuerpo. La cabecera contiene información *sobre* el documento, y el cuerpo contiene el documento en sí. En este caso, la cabecera declara que el título de este documento es “Mi página de inicio” y que utiliza la codificación UTF-8, que es una forma de codificar texto Unicode como datos binarios. El cuerpo del documento contiene un encabezado (<h1>, que significa “encabezado 1” — <h2> a <h6> producen subencabezados) y dos párrafos (<p>).

Las etiquetas vienen en varias formas. Un elemento, como el cuerpo, un párrafo o un enlace, comienza con una *etiqueta de apertura* como <p> y finaliza con una *etiqueta de cierre* como </p>. Algunas etiquetas de apertura, como la de enlace (<a>), contienen información adicional en forma de pares nombre="valor". Estos se llaman *atributos*. En este caso, el destino del enlace se indica con

`href="http://eloquentjavascript.net"`, donde `href` significa “hipervínculo de referencia”.

Algunos tipos de etiquetas no contienen nada y por lo tanto no necesitan ser cerradas. La etiqueta de metadatos `<meta charset="utf-8">` es un ejemplo de esto.

Para poder incluir corchetes angulares en el texto de un documento, a pesar de que tienen un significado especial en HTML, se debe introducir otra forma especial de notación. Un simple signo menor que se escribe como `<` (“menor que”), y un signo mayor que se escribe como `>` (“mayor que”). En HTML, un carácter y comercial (&) seguido de un nombre o código de carácter y un punto y coma (;) se llama una *entidad* y será reemplazado por el carácter que codifica.

Esto es análogo a la manera en que se utilizan las barras invertidas en las cadenas de texto de JavaScript. Dado que este mecanismo también otorga un significado especial a los caracteres de y comercial, necesitan ser escapados como `&`. Dentro de los valores de los atributos, que están entre comillas dobles, se puede usar `"` para insertar un carácter de comillas real.

HTML se analiza de una manera notablemente tolerante a errores. Cuando faltan etiquetas que deberían estar ahí, el navegador las agrega automáticamente. La forma en que se hace esto se ha estandarizado, y puedes confiar en que todos los navegadores modernos lo harán de la misma manera.

El siguiente documento será tratado igual que el que se mostró anteriormente:

```
<!doctype html>

<meta charset=utf-8>
<title>Mi página de inicio</title>

<h1>Mi página de inicio</h1>
<p>Hola, soy Marijn y esta es mi página de inicio.
<p>También escribí un libro! Léelo
  <a href=http://eloquentjavascript.net>aquí</a>.
```

Las etiquetas `<html>`, `<head>` y `<body>` han desaparecido por completo. El navegador sabe que `<meta>` y `<title>` pertenecen a la cabecera y que `<h1>` significa que el cuerpo ha comenzado. Además, ya no cierro explícitamente los párrafos, ya que abrir un nuevo párrafo o finalizar el documento los cerrará implícitamente. Las comillas alrededor de los valores de los atributos también han desaparecido.

Este libro generalmente omitirá las etiquetas `<html>`, `<head>` y `<body>` en ejemplos para mantenerlos cortos y libres de desorden. Pero sí cerraré las etiquetas e incluiré comillas alrededor de los atributos.

También generalmente omitiré el doctype y la declaración `charset`. Esto no debe interpretarse como una recomendación para omitirlos de documentos HTML. Los navegadores a menudo hacen cosas ridículas cuando los olvidas. Deberías considerar que el doctype y los metadatos del `charset` están implícitamente presentes en los ejemplos, incluso cuando no se muestran realmente en el texto.

HTML Y JAVASCRIPT

En el contexto de este libro, la etiqueta HTML más importante es `<script>`. Esta etiqueta nos permite incluir un fragmento de JavaScript en un documento.

```
<h1>Probando alerta</h1>
<script>alert("¡hola!");</script>
```

Dicho script se ejecutará tan pronto como su etiqueta `<script>` sea encontrada mientras el navegador lee el HTML. Esta página mostrará un cuadro de diálogo al abrirla—la función `alert` se asemeja a `prompt`, en que muestra una ventana pequeña, pero solo muestra un mensaje sin solicitar entrada.

Incluir programas extensos directamente en documentos HTML a menudo es poco práctico. La etiqueta `<script>` puede recibir un atributo `src` para obtener un archivo de script (un archivo de texto que contiene un programa JavaScript) desde una URL.

```
<h1>Probando alerta</h1>
<script src="code/hello.js"></script>
```

El archivo *code/hello.js* incluido aquí contiene el mismo programa—`alert("¡hola!")`. Cuando una página HTML referencia otras URL como parte de sí misma—por ejemplo, un archivo de imagen o un script—los navegadores web los recuperarán inmediatamente e incluirán en la página.

Una etiqueta de script siempre debe cerrarse con `</script>`, incluso si hace referencia a un archivo de script y no contiene ningún código. Si olvidas esto, el resto de la página se interpretará como parte del script.

Puedes cargar módulos ES (ver [Capítulo 10](#)) en el navegador al darle a tu etiqueta de script un atributo `type="module"`. Dichos módulos pueden depender de otros módulos usando URLs relativas a sí mismos como nombres de módulo en declaraciones de `import`.

Algunos atributos también pueden contener un programa JavaScript. La etiqueta `<button>` (que se muestra como un botón) soporta un atributo `onclick`. El valor del atributo se ejecutará cada vez que se haga clic en el botón.

```
<button onclick="alert('¡Boom!');">¡NO PRESIONES!</button>
```

Nota que tuve que utilizar comillas simples para el string en el atributo `onclick` porque las comillas dobles ya se usan para citar todo el atributo. También podría haber utilizado `"` ; .

EN EL ENTORNO CONTROLADO

Ejecutar programas descargados de Internet es potencialmente peligroso. No sabes mucho sobre las personas detrás de la mayoría de los sitios que visitas, y no necesariamente tienen buenas intenciones. Ejecutar programas de personas que no tienen buenas intenciones es cómo se infecta tu computadora con virus, te roban tus datos y hackean tus cuentas.

Sin embargo, la atracción de la Web es que puedes navegar por ella sin necesariamente confiar en todas las páginas que visitas. Por eso, los navegadores limitan severamente las cosas que un programa JavaScript puede hacer: no puede ver los archivos en tu computadora ni modificar nada que no esté relacionado con la página web en la que estaba incrustado.

Aislar un entorno de programación de esta manera se llama *sandboxing*, la idea es que el programa está jugando inofensivamente en un arenero. Pero debes imaginar este tipo particular de arenero como teniendo una jaula de barras de acero gruesas sobre él para que los programas que juegan en él no puedan salir realmente.

La parte difícil del sandboxing es permitir que los programas tengan suficiente espacio para ser útiles y al mismo tiempo restringirlos para que no hagan nada peligroso. Muchas funcionalidades útiles, como comunicarse con otros servidores o leer el contenido del portapapeles, también pueden usarse para hacer cosas problemáticas que invaden la privacidad.

De vez en cuando, alguien encuentra una nueva forma de evitar las limitaciones de un navegador y hacer algo dañino, que va desde filtrar información privada menor hasta tomar el control de toda la máquina en la que se ejecuta el navegador. Los desarrolladores de navegadores responden reparando el agujero, y todo vuelve a estar bien, hasta que se descubre el próximo problema, y con suerte se publicita, en lugar de ser explotado en secreto por alguna agencia gubernamental u organización criminal.

COMPATIBILIDAD Y LAS GUERRAS DE NAVEGADORES

En las etapas iniciales de la Web, un navegador llamado Mosaic dominaba el mercado. Después de unos años, el equilibrio se desplazó a Netscape, que a su vez fue en gran medida reemplazado por Internet Explorer de Microsoft. En cualquier punto en el que un

único navegador era dominante, el fabricante de ese navegador se creía con derecho a inventar nuevas funciones para la Web unilateralmente. Dado que la mayoría de usuarios usaban el navegador más popular, los sitio webs simplemente comenzaban a usar esas características, sin importar los otros navegadores.

Esta fue la era oscura de la compatibilidad, a menudo llamada las *guerras de navegadores*. Los desarrolladores web se quedaron con no una Web unificada, sino dos o tres plataformas incompatibles. Para empeorar las cosas, los navegadores en uso alrededor de 2003 estaban llenos de errores, y por supuesto los errores eran diferentes para cada navegador. La vida era difícil para las personas que escribían páginas web.

Mozilla Firefox, un derivado sin ánimo de lucro de Netscape, desafió la posición de Internet Explorer a finales de la década de 2000. Debido a que Microsoft no estaba particularmente interesado en mantenerse competitivo en ese momento, Firefox le quitó mucho cuota de mercado. Alrededor del mismo tiempo, Google introdujo su navegador Chrome y el navegador de Apple Safari ganó popularidad, lo que llevó a una situación en la que había cuatro actores principales, en lugar de uno solo.

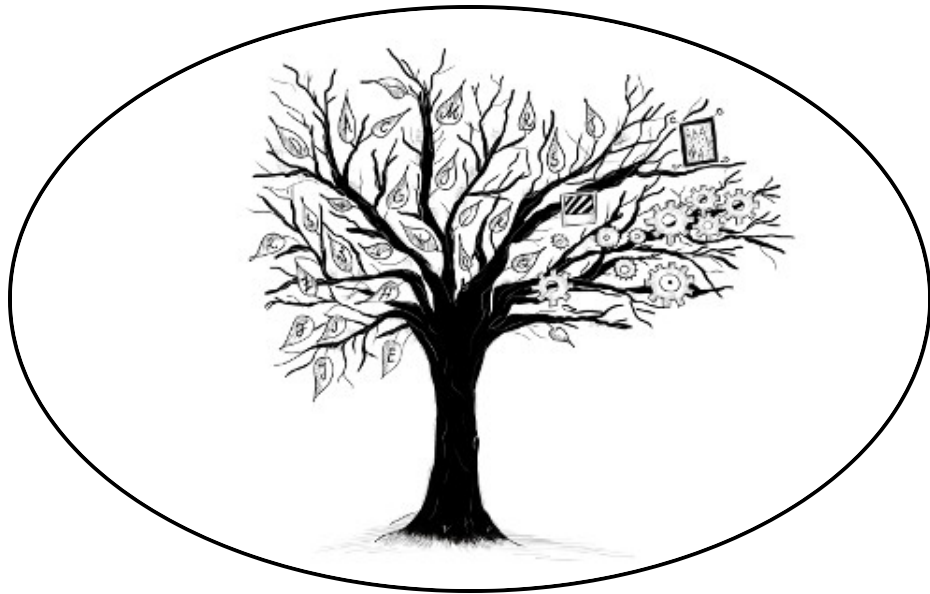
Los nuevos actores tenían una actitud más seria hacia los estándares y mejores prácticas de ingeniería, lo que nos dio menos incompatibilidad y menos errores. Microsoft, viendo cómo su cuota de mercado se desmoronaba, adoptó estas actitudes en su navegador Edge, que reemplaza a Internet Explorer. Si estás empezando a aprender desarrollo web hoy, considérate afortunado. Las últimas versiones de los principales navegadores se comportan de manera

bastante uniforme y tienen relativamente pocos errores. Desafortunadamente, con la disminución constante de la cuota de mercado de Firefox y Edge convirtiéndose en simplemente un contenedor alrededor del núcleo de Chrome en 2018, esta uniformidad podría una vez más tomar la forma de un único proveedor —Google en este caso— teniendo el suficiente control sobre el mercado de navegadores para imponer su idea de cómo debería lucir la Web al resto del mundo.

EL MODELO DE OBJETOS DEL DOCUMENTO

“¡Qué mal! ¡La misma vieja historia! Una vez que has terminado de construir tu casa, te das cuenta de que has aprendido accidentalmente algo que realmente deberías haber sabido antes de comenzar.”

—Friedrich Nietzsche, *Más allá del bien y del mal*



Cuando abres una página web, tu navegador recupera el texto HTML de la página y lo analiza, de manera similar a como nuestro analizador de [Capítulo 12](#) analizaba programas. El navegador construye un modelo de la estructura del documento y utiliza este modelo para dibujar la página en la pantalla.

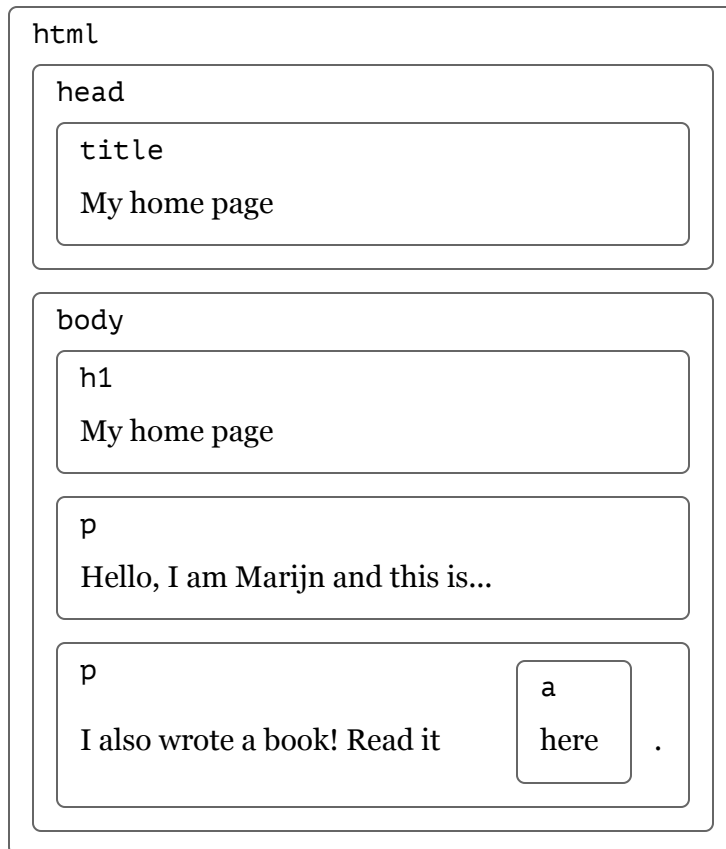
Esta representación del documento es uno de los juguetes que un programa JavaScript tiene disponible en su caja de arena. Es una estructura de datos que puedes leer o modificar. Actúa como una estructura de datos *en vivo*: cuando se modifica, la página en la pantalla se actualiza para reflejar los cambios.

ESTRUCTURA DEL DOCUMENTO

Puedes imaginar un documento HTML como un conjunto anidado de cajas. Etiquetas como `<body>` y `</body>` encierran otras etiquetas, que a su vez contienen otras etiquetas o texto. Aquí está el documento de ejemplo del [capítulo anterior](#):

```
<!doctype html>
<html>
  <head>
    <title>Mi página de inicio</title>
  </head>
  <body>
    <h1>Mi página de inicio</h1>
    <p>Hola, soy Marijn y esta es mi página de inicio.</p>
    <p>¡También escribí un libro! Léelo
      <a href="http://eloquentjavascript.net">aquí</a>.</p>
  </body>
</html>
```

Esta página tiene la siguiente estructura:



La estructura de datos que el navegador utiliza para representar el documento sigue esta forma. Para cada caja, hay un objeto con el que podemos interactuar para saber cosas como qué etiqueta HTML representa y qué cajas y texto contiene. Esta representación se llama *Modelo de Objetos del Documento*, o DOM en resumen.

El enlace `global document` nos da acceso a estos objetos. Su propiedad `documentElement` se refiere al objeto que representa la etiqueta `<html>`. Dado que cada documento HTML tiene una cabeza y un cuerpo, también tiene propiedades `head` y `body`, que apuntan a esos elementos.

ÁRBOLES

Piensa en los árbol sintácticos del [Capítulo 12](#) por un momento. Sus estructuras son sorprendentemente similares a la estructura de un documento de un navegador. Cada *nodo* puede referirse a otros nodos, *hijos*, que a su vez pueden tener sus propios hijos. Esta forma es típica de estructuras anidadas donde los elementos pueden contener subelementos que son similares a ellos mismos.

Llamamos a una estructura de datos un *árbol* cuando tiene una estructura de ramificación, no tiene ciclos (un nodo no puede contenerse a sí mismo, directa o indirectamente), y tiene un *raíz* única y bien definida. En el caso del DOM, `document` . `documentElement` sirve como la raíz.

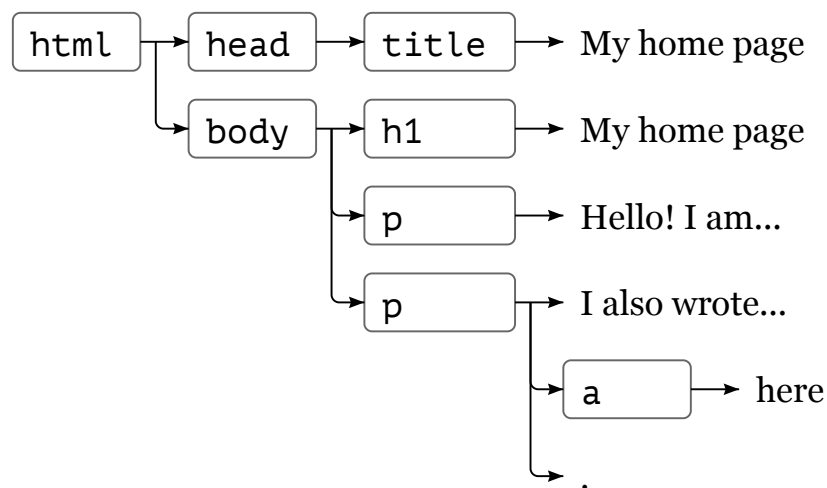
Los árboles son comunes en la informática. Además de representar estructuras recursivas como documentos HTML o programas, a menudo se utilizan para mantener conjuntos de datos ordenados porque los elementos generalmente se pueden encontrar o insertar de manera más eficiente en un árbol que en un arreglo plano.

Un árbol típico tiene diferentes tipos de nodos. El árbol de sintaxis para [el lenguaje Egg](#) tenía identificadores, valores y nodos de aplicación. Los nodos de aplicación pueden tener hijos, mientras que los identificadores y valores son *hojas*, o nodos sin hijos.

Lo mismo ocurre para el DOM. Los nodos de los *elementos*, que representan etiquetas HTML, determinan la estructura del documento. Estos pueden tener nodo hijos. Un ejemplo de dicho nodo es `document . body` . Algunos de estos hijos pueden ser nodo hoja, como fragmentos de texto o nodos comentario.

Cada objeto de nodo del DOM tiene una propiedad `nodeType`, que contiene un código (número) que identifica el tipo de nodo. Los elementos tienen el código 1, que también se define como la propiedad constante `Node.ELEMENT_NODE`. Los nodos de texto, que representan una sección de texto en el documento, obtienen el código 3 (`Node.TEXT_NODE`). Los comentarios tienen el código 8 (`Node.COMMENT_NODE`).

Otra forma de visualizar nuestro árbol de documento es la siguiente:



Las hojas son nodos de texto, y las flechas indican las relaciones padre-hijo entre nodos.

EL ESTÁNDAR

Usar códigos numéricos crípticos para representar tipos de nodos no es algo muy propio de JavaScript. Más adelante en este capítulo, veremos que otras partes de la interfaz del DOM también se sienten incómodas y extrañas. La razón de esto es que la interfaz del DOM no fue diseñada exclusivamente para JavaScript. Más bien, intenta ser una interfaz neutral en cuanto a lenguaje que también pueda

utilizarse en otros sistemas, no solo para HTML, sino también para XML, que es un formato de datos genérico con una sintaxis similar a HTML.

Esto es lamentable. Los estándares a menudo son útiles. Pero en este caso, la ventaja (consistencia entre lenguajes) no es tan convincente. Tener una interfaz que esté correctamente integrada con el lenguaje que estás utilizando te ahorrará más tiempo que tener una interfaz familiar en varios lenguajes.

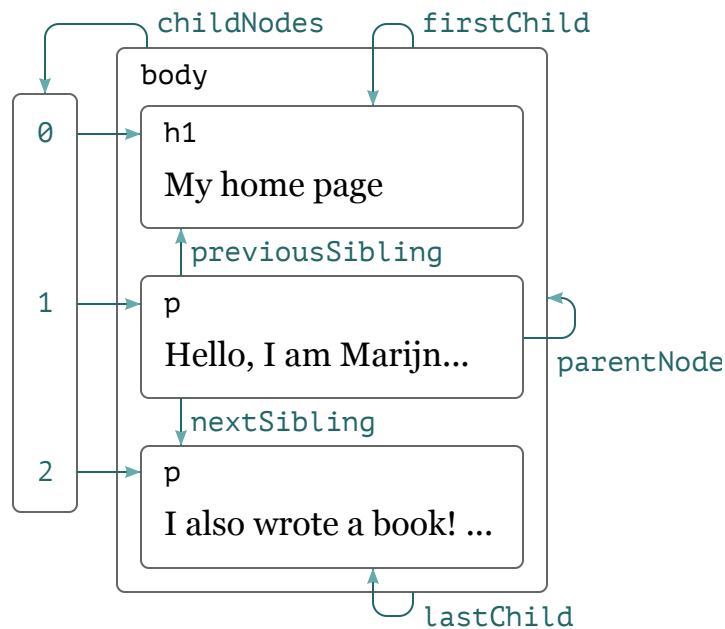
Como ejemplo de esta mala integración, considera la propiedad `childNodes` que tienen los nodos de elementos en el DOM. Esta propiedad contiene un objeto similar a un array, con una propiedad `length` y propiedades etiquetadas por números para acceder a los nodos hijos. Pero es una instancia del tipo `NodeList`, no un array real, por lo que no tiene métodos como `slice` y `map`.

Luego, hay problemas que son simplemente de mala diseño. Por ejemplo, no hay forma de crear un nuevo nodo y agregar inmediatamente hijos o atributos a él. En su lugar, primero tienes que crearlo y luego agregar los hijos y atributos uno por uno, usando efectos secundarios. El código que interactúa mucho con el DOM tiende a ser largo, repetitivo y feo.

Pero estos defectos no son fatales. Dado que JavaScript nos permite crear nuestras propias abstracciones, es posible diseñar formas mejoradas de expresar las operaciones que estás realizando. Muchas bibliotecas destinadas a la programación del navegador vienen con herramientas de este tipo.

MOVIMIENTO A TRAVÉS DEL ÁRBOL

Los nodos DOM contienen una gran cantidad de enlaces a otros nodos cercanos. El siguiente diagrama ilustra esto:



Aunque el diagrama muestra solo un enlace de cada tipo, cada nodo tiene una propiedad `parentNode` que apunta al nodo del que forma parte, si lo hay. De igual manera, cada nodo de elemento (tipo 1) tiene una propiedad `childNodes` que apunta a un objeto similar a un array que contiene sus hijos.

En teoría, podrías moverte por todo el árbol utilizando solo estos enlaces padre e hijo. Pero JavaScript también te da acceso a varios enlaces de conveniencia adicionales. Las propiedades `firstChild` y `lastChild` apuntan a los primeros y últimos elementos hijos o tienen el valor `null` para nodos sin hijos. De manera similar, `previousSibling` y `nextSibling` apuntan a nodos adyacentes, que son nodos con el mismo padre que aparecen inmediatamente antes o después del nodo en sí. Para un primer hijo,

`previousSibling` será nulo, y para un último hijo, `nextSibling` será nulo.

También está la propiedad `children`, que es como `childNodes` pero contiene solo hijos de elementos (tipo 1), no otros tipos de nodos hijos. Esto puede ser útil cuando no estás interesado en nodos de texto.

Cuando se trabaja con una estructura de datos anidada como esta, las funciones recursivas son frecuentemente útiles. La siguiente función examina un documento en busca de nodos de texto que contengan una cadena específica y devuelve `true` cuando ha encontrado uno:

```
function talksAbout(node, cadena) {
  if (node.nodeType == Node.ELEMENT_NODE) {
    for (let child of node.childNodes) {
      if (talksAbout(child, cadena)) {
        return true;
      }
    }
    return false;
  } else if (node.nodeType == Node.TEXT_NODE) {
    return node.nodeValue.indexOf(cadena) > -1;
  }
}

console.log(talksAbout(document.body, "libro"));
// → true
```

La propiedad `nodeValue` de un nodo de texto contiene la cadena de texto que representa.

ENCONTRANDO ELEMENTOS

Navegar por estos enlaces entre padres, hijos y hermanos a menudo es útil. Pero si queremos encontrar un nodo específico en el documento, llegar a él empezando por `document.body` y siguiendo un camino fijo de propiedades no es una buena idea. Hacerlo implica hacer suposiciones en nuestro programa sobre la estructura precisa del documento, una estructura que podrías querer cambiar más adelante. Otro factor complicador es que se crean nodos de texto incluso para los espacios en blanco entre nodos. La etiqueta `<body>` del documento de ejemplo no tiene solo tres hijos (`<h1>` y dos elementos `<p>`) sino que en realidad tiene siete: esos tres, más los espacios en blanco antes, después y entre ellos.

Por lo tanto, si queremos obtener el atributo `href` del enlace en ese documento, no queremos decir algo como “Obtener el segundo hijo del sexto hijo del cuerpo del documento”. Sería mejor si pudiéramos decir “Obtener el primer enlace en el documento”. Y podemos hacerlo.

```
let enlace = document.body.getElementsByTagName("a")[0];
console.log(enlace.href);
```

Todos los nodos de elemento tienen un método `getElementsByTagName`, que recoge todos los elementos con el nombre de etiqueta dado que son descendientes (hijos directos o indirectos) de ese nodo y los devuelve como un objeto similar a un array.

Para encontrar un nodo específico *único*, puedes darle un atributo `id` y usar `document.getElementById` en su lugar.

```
<p>Mi avestruz Gertrudis:</p>
<p></p>
```

```
<script>
  let ostrich = document.getElementById("gertrudis");
  console.log(ostrich.src);
</script>
```

Un tercer método similar es `getElementsByClassName`, que, al igual que `getElementsByTagName`, busca a través del contenido de un nodo de elemento y recupera todos los elementos que tienen la cadena dada en su atributo `class`.

CAMBIANDO EL DOCUMENTO

Casi todo se puede cambiar en la estructura de datos del DOM. La forma del árbol del documento se puede modificar cambiando las relaciones padre-hijo. Los nodos tienen un método `remove` para removerlos de su nodo padre actual. Para añadir un nodo hijo a un nodo de elemento, podemos usar `appendChild`, que lo coloca al final de la lista de hijos, o `insertBefore`, que inserta el nodo dado como primer argumento antes del nodo dado como segundo argumento.

```
<p>Uno</p>
<p>Dos</p>
<p>Tres</p>

<script>
  let párrafos = document.body.getElementsByTagName("p");
  document.body.insertBefore(párrafos[2], párrafos[0]);
</script>
```

Un nodo puede existir en el documento en un solo lugar. Por lo tanto, insertar el párrafo *Tres* delante del párrafo *Uno* primero lo removerá del final del documento y luego lo insertará al principio,

resultando en *Tres/Uno/Dos*. Todas las operaciones que insertan un nodo en algún lugar causarán, como un efecto secundario, que se elimine de su posición actual (si tiene una).

El método `replaceChild` se usa para reemplazar un nodo hijo con otro. Toma como argumentos dos nodos: un nodo nuevo y el nodo que se reemplazará. El nodo reemplazado debe ser un hijo del elemento en el que se llama el método. Ten en cuenta que tanto `replaceChild` como `insertBefore` esperan que el nodo *nuevo* sea su primer argumento.

CREACIÓN DE NODOS

Digamos que queremos escribir un script que reemplace todas las imágenes (etiquetas ``) en el documento con el texto contenido en sus atributos `alt`, que especifica una representación textual alternativa de la imagen.

Esto implica no solo eliminar las imágenes sino agregar un nuevo nodo de texto para reemplazarlas.

```
<p>The  in the  
  .</p>
```

```
<p><button onclick="replaceImages()">Replace</button></p>
```

```
<script>  
  function replaceImages() {  
    let images = document.getElementsByTagName("img");  
    for (let i = images.length - 1; i >= 0; i--) {  
      let image = images[i];  
      if (image.alt) {  
        let text = document.createTextNode(image.alt);  
        image.parentNode.replaceChild(text, image);  
      }  
    }  
  }  
</script>
```

```
    }  
  }  
</script>
```

Dada una cadena, `createTextNode` nos da un nodo de texto que podemos insertar en el documento para que aparezca en la pantalla.

El bucle que recorre las imágenes comienza al final de la lista. Esto es necesario porque la lista de nodos devuelta por un método como `getElementsByTagName` (o una propiedad como `childNodes`) es *dinámica*. Es decir, se actualiza a medida que el documento cambia. Si comenzáramos desde el principio, al quitar la primera imagen haría que la lista perdiera su primer elemento, por lo que la segunda vez que se repita el bucle, cuando `i` es 1, se detendría porque la longitud de la colección ahora también es 1.

Si quieres tener una colección *sólida* de nodos, en lugar de una en vivo, puedes convertir la colección en un array real llamando a `Array.from`.

```
let arrayish = {0: "uno", 1: "dos", length: 2};  
let array = Array.from(arrayish);  
console.log(array.map(s => s.toUpperCase()));  
// → ["UNO", "DOS"]
```

Para crear nodos elemento, puedes utilizar el método `document.createElement`. Este método toma un nombre de etiqueta y devuelve un nuevo nodo vacío del tipo dado.

El siguiente ejemplo define una utilidad `elt`, que crea un nodo de elemento y trata el resto de sus argumentos como hijos de ese nodo. Luego, esta función se utiliza para agregar una atribución a una cita.

```

<blockquote id="quote">
  Ningún libro puede considerarse terminado. Mientras trabajamos
  en él aprendemos
  lo suficiente como para encontrarlo inmaduro en el momento en
  que lo dejamos.
</blockquote>

<script>
  function elt(type, ...children) {
    let node = document.createElement(type);
    for (let child of children) {
      if (typeof child !== "string") node.appendChild(child);
      else node.appendChild(document.createTextNode(child));
    }
    return node;
  }

  document.getElementById("quote").appendChild(
    elt("footer", "-",
      elt("strong", "Karl Popper"),
      ", prefacio de la segunda edición de ",
      elt("em", "La sociedad abierta y sus enemigos"),
      ", 1950"));
</script>

```

Así es como se vería el documento resultante:

No book can ever be finished. While working on it we learn
just enough to find it immature the moment we turn away
from it.
—**Karl Popper**, preface to the second editon of *The Open
Society and Its Enemies*, 1950

ATRIBUTOS

Algunos atributos de elementos, como `href` para enlaces, pueden ser accedidos a través de una propiedad con el mismo nombre en el

objeto DOM del elemento. Este es el caso para la mayoría de atributos estándar comúnmente usados.

HTML te permite establecer cualquier atributo que desees en los nodos. Esto puede ser útil porque te permite almacenar información adicional en un documento. Para leer o cambiar atributos personalizados, que no están disponibles como propiedades regulares del objeto, debes usar los métodos `getAttribute` y `setAttribute`.

```
<p data-classified="secreto">El código de lanzamiento es 00000000.
</p>
<p data-classified="no clasificado">Tengo dos pies.</p>

<script>
  let paras = document.getElementsByTagName("p");
  for (let para of Array.from(paras)) {
    if (para.getAttribute("data-classified") == "secreto") {
      para.remove();
    }
  }
</script>
```

Se recomienda prefijar los nombres de estos atributos inventados con `data-` para asegurarse de que no entren en conflicto con otros atributos.

Existe un atributo comúnmente usado, `class`, que es una palabra clave en el lenguaje JavaScript. Por razones históricas—algunas implementaciones antiguas de JavaScript no podían manejar nombres de propiedades que coincidieran con palabras clave—la propiedad utilizada para acceder a este atributo se llama `className`. También puedes acceder a él con su nombre real,

"class", utilizando los métodos `getAttribute` y `setAttribute`.

DISEÑO

Puede que hayas notado que diferentes tipos de elementos se disponen de manera diferente. Algunos, como párrafos (`<p>`) o encabezados (`<h1>`), ocupan todo el ancho del documento y se muestran en líneas separadas. Estos se llaman elementos de *bloque*. Otros, como enlaces (`<a>`) o el elemento ``, se muestran en la misma línea que el texto que los rodea. A estos elementos se les llama elementos *en línea*.

Para cualquier documento dado, los navegadores son capaces de calcular un diseño, que le da a cada elemento un tamaño y posición basados en su tipo y contenido. Luego, este diseño se usa para dibujar el documento realmente.

El tamaño y posición de un elemento pueden ser accedidos desde JavaScript. Las propiedades `offsetWidth` y `offsetHeight` te dan el espacio que el elemento ocupa en *píxeles*. Un píxel es la unidad básica de medida en el navegador. Tradicionalmente corresponde al punto más pequeño que la pantalla puede dibujar, pero en pantallas modernas, que pueden dibujar puntos *muy* pequeños, eso puede que ya no sea cierto, y un píxel del navegador puede abarcar múltiples puntos de la pantalla.

De manera similar, `clientWidth` y `clientHeight` te dan el tamaño del espacio *dentro* del elemento, ignorando el ancho del borde.

```
<p style="border: 3px solid red">
  Estoy enmarcado
</p>

<script>
  let para = document.getElementsByTagName("p")[0];
  console.log("clientHeight:", para.clientHeight);
  // → 19
  console.log("offsetHeight:", para.offsetHeight);
  // → 25
</script>
```

Darle a un párrafo un borde hace que se dibuje un rectángulo a su alrededor.

I'm boxed in

La manera más efectiva de encontrar la posición precisa de un elemento en la pantalla es el método `getBoundingClientRect`. Devuelve un objeto con las propiedades `top`, `bottom`, `left` y `right`, indicando las posiciones en píxeles de los lados del elemento en relación con la esquina superior izquierda de la pantalla. Si los quieres en relación al documento completo, debes sumar la posición actual de desplazamiento, que puedes encontrar en las variables `pageXOffset` y `pageYOffset`.

Diseñar un documento puede ser bastante trabajo. En aras de la rapidez, los motores de los navegadores no vuelven a diseñar inmediatamente un documento cada vez que se modifica, sino que esperan tanto como pueden. Cuando un programa de JavaScript que ha modificado el documento finaliza su ejecución, el navegador tendrá que calcular un nuevo diseño para dibujar el documento modificado en la pantalla. Cuando un programa *pide* la posición o

tamaño de algo leyendo propiedades como `offsetHeight` o llamando a `getBoundingClientRect`, proporcionar esa información también requiere calcular un diseño.

Un programa que alterna repetidamente entre la lectura de información de diseño del DOM y el cambio del DOM provoca que se realicen muchas computaciones de diseño y, en consecuencia, se ejecute muy lentamente. El siguiente código es un ejemplo de esto. Contiene dos programas diferentes que construyen una línea de caracteres *X* de 2,000 píxeles de ancho y mide el tiempo que lleva cada uno.

```
<p><span id="one"></span></p>
<p><span id="two"></span></p>

<script>
  function time(name, action) {
    let start = Date.now(); // Tiempo actual en milisegundos
    action();
    console.log(name, "tomó", Date.now() - start, "ms");
  }

  time("ingenuo", () => {
    let target = document.getElementById("one");
    while (target.offsetWidth < 2000) {
      target.appendChild(document.createTextNode("X"));
    }
  });
  // → ingenuo tomó 32 ms

  time("astuto", function() {
    let target = document.getElementById("two");
    target.appendChild(document.createTextNode("XXXXX"));
    let total = Math.ceil(2000 / (target.offsetWidth / 5));
    target.firstChild.nodeValue = "X".repeat(total);
  });
  // → astuto tomó 1 ms
</script>
```

ESTILOS

Hemos visto que diferentes elementos HTML se dibujan de manera diferente. Algunos se muestran como bloques, otros en línea. Algunos agregan estilos: `` hace que su contenido sea negrita, y `<a>` lo hace azul y lo subraya.

La forma en que una etiqueta `` muestra una imagen o una etiqueta `<a>` hace que se siga un enlace al hacer clic está fuertemente vinculada al tipo de elemento. Pero podemos cambiar el estilo asociado con un elemento, como el color del texto o el subrayado. Aquí hay un ejemplo que utiliza la propiedad `style`:

```
<p><a href=".">Enlace normal</a></p>
<p><a href="." style="color: green">Enlace verde</a></p>
```

El segundo enlace será verde en lugar del color de enlace predeterminado.

[Normal link](#)

[Green link](#)

Un atributo de estilo puede contener uno o más *declaraciones*, que son una propiedad (como `color`) seguida de dos puntos y un valor (como `verde`). Cuando hay más de una declaración, deben separarse por punto y comas, como en `"color: rojo; border: ninguno"`.

Muchos aspectos del documento pueden ser influenciados por el estilo. Por ejemplo, la propiedad `display` controla si un elemento se muestra como un bloque o como un elemento en línea.

```
Este texto se muestra de forma <strong>en línea</strong>,  
<strong style="display: block">como un bloque</strong>, y  
<strong style="display: none">no del todo</strong>.
```

La etiqueta `block` terminará en su propia línea ya que los elementos de bloque no se muestran en línea con el texto que los rodea. La última etiqueta no se muestra en absoluto: `display: none` evita que un elemento aparezca en la pantalla. Esta es una forma de ocultar elementos. A menudo es preferible a eliminarlos completamente del documento porque facilita revelarlos nuevamente más tarde.

This text is displayed **inline**,
as a block
, and .

El código JavaScript puede manipular directamente el estilo de un elemento a través de la propiedad `style` del elemento. Esta propiedad contiene un objeto que tiene propiedades para todas las posibles propiedades de estilo. Los valores de estas propiedades son cadenas de texto, a las cuales podemos escribir para cambiar un aspecto particular del estilo del elemento.

```
<p id="para" style="color: purple">  
  Texto bonito  
</p>
```

```
<script>  
  let para = document.getElementById("para");
```

```
console.log(para.style.color);
para.style.color = "magenta";
</script>
```

Algunos nombres de propiedades de estilo contienen guiones, como `font-family`. Debido a que trabajar con estos nombres de propiedades en JavaScript es incómodo (tendrías que decir `style["font-family"]`), los nombres de las propiedades en el objeto `style` para tales propiedades tienen los guiones eliminados y las letras posterior a ellos en mayúscula (`style.fontFamily`).

ESTILOS EN CASCADA

El sistema de estilos para HTML se llama CSS, por sus siglas en inglés, *Cascading Style Sheets*. Una *hoja de estilo* es un conjunto de reglas sobre cómo dar estilo a los elementos en un documento. Puede ser proporcionada dentro de una etiqueta `<style>`.

```
<style>
  strong {
    font-style: italic;
    color: gray;
  }
</style>
<p>Ahora el <strong>texto fuerte</strong> es cursiva y gris.</p>
```

El *cascada* en el nombre se refiere al hecho de que múltiples reglas de este tipo se combinan para producir el estilo final de un elemento. En el ejemplo, el estilo predeterminado de las etiquetas ``, que les da `font-weight: bold`, se superpone por la regla en la etiqueta `<style>`, que agrega `font-style` y `color`.

Cuando múltiples reglas definen un valor para la misma propiedad, la regla más recientemente leída obtiene una precedencia más alta y

gana. Por lo tanto, si la regla en la etiqueta `<style>` incluyera `font-weight: normal`, contradiciendo la regla predeterminada de `font-weight`, el texto sería normal, *no* negrita. Los estilos en un atributo `style` aplicado directamente al nodo tienen la mayor precedencia y siempre prevalecen.

Es posible apuntar a cosas distintas de los nombres de etiqueta en reglas de CSS. Una regla para `.abc` se aplica a todos los elementos con `"abc"` en su atributo `class`. Una regla para `#xyz` se aplica al elemento con un atributo `id` de `"xyz"` (que debería ser único dentro del documento).

```
.subtle {
  color: gray;
  font-size: 80%;
}
#header {
  background: blue;
  color: white;
}
/* elementos p con id main y con clases a y b */
p#main.a.b {
  margin-bottom: 20px;
}
```

La regla de precedencia que favorece a la regla más recientemente definida se aplica solo cuando las reglas tienen la misma *especificidad*. La especificidad de una regla es una medida de qué tan precisamente describe los elementos que coinciden, determinada por el número y tipo (etiqueta, clase o ID) de aspectos de elementos que requiere. Por ejemplo, una regla que apunta a `p.a` es más específica que las reglas que apuntan a `p` o simplemente `.a` y, por lo tanto, tendría precedencia sobre ellas.

La notación `p > a {...}` aplica los estilos dados a todas las etiquetas `<a>` que son hijos directos de etiquetas `<p>`. De manera similar, `p a {...}` se aplica a todas las etiquetas `<a>` dentro de las etiquetas `<p>`, ya sean hijos directos o indirectos.

SELECTORES DE CONSULTA

No vamos a usar hojas de estilo demasiado en este libro. Entenderlas es útil cuando se programa en el navegador, pero son lo suficientemente complicadas como para justificar un libro aparte.

La razón principal por la que introduje la sintaxis *selector*—la notación utilizada en las hojas de estilo para determinar a qué elementos se aplican un conjunto de estilos— es que podemos utilizar este mismo mini-lenguaje como una forma efectiva de encontrar elementos del DOM.

El método `querySelectorAll`, que está definido tanto en el objeto `document` como en los nodos de elementos, toma una cadena de selector y devuelve un `NodeList` que contiene todos los elementos que encuentra.

```
<p>And if you go chasing
  <span class="animal">rabbits</span></p>
<p>And you know you're going to fall</p>
<p>Tell 'em a <span class="character">hookah smoking
  <span class="animal">caterpillar</span></span></p>
<p>Has given you the call</p>

<script>
  function count(selector) {
    return document.querySelectorAll(selector).length;
  }
  console.log(count("p"));           // Todos los elementos <p>
```

```
// → 4
console.log(count(".animal"));      // Clase animal
// → 2
console.log(count("p .animal"));    // Animal dentro de <p>
// → 2
console.log(count("p > .animal"));  // Hijo directo de <p>
// → 1
</script>
```

A diferencia de métodos como `getElementsByTagName`, el objeto devuelto por `querySelectorAll` *no* es dinámico. No cambiará cuando cambies el documento. Aun así, no es un array real, por lo que necesitas llamar a `Array.from` si deseas tratarlo como tal.

El método `querySelector` (sin la parte `All`) funciona de manera similar. Este es útil si deseas un elemento específico y único. Solo devolverá el primer elemento coincidente o `null` cuando no haya ningún elemento coincidente.

POSICIONAMIENTO Y ANIMACIÓN

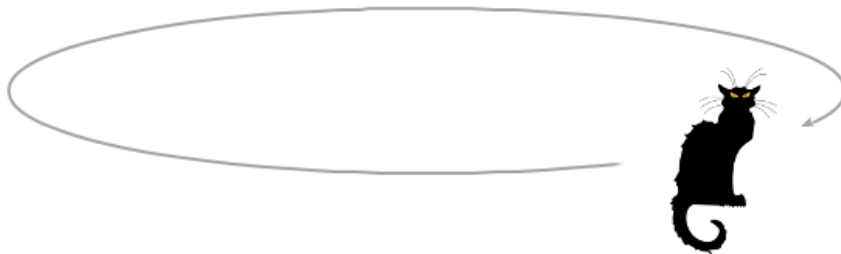
La propiedad de estilo `position` influye en el diseño de una manera poderosa. De forma predeterminada, tiene un valor de `static`, lo que significa que el elemento se sitúa en su lugar normal en el documento. Cuando se establece en `relative`, el elemento sigue ocupando espacio en el documento, pero ahora las propiedades de estilo `top` y `left` se pueden usar para moverlo con respecto a ese lugar normal. Cuando `position` se establece en `absolute`, el elemento se elimina del flujo normal del documento, es decir, ya no ocupa espacio y puede superponerse con otros elementos. Además, sus propiedades de `top` y `left` se pueden usar para posicionarlo absolutamente con respecto a la esquina superior izquierda del elemento contenedor más cercano cuya propiedad de `position` no

sea `static`, o con respecto al documento si no existe tal elemento contenedor.

Podemos usar esto para crear una animación. El siguiente documento muestra una imagen de un gato que se mueve en una elipse:

```
<p style="text-align: center">
  
</p>
<script>
  let cat = document.querySelector("img");
  let angle = Math.PI / 2;
  function animate(time, lastTime) {
    if (lastTime != null) {
      angle += (time - lastTime) * 0.001;
    }
    cat.style.top = (Math.sin(angle) * 20) + "px";
    cat.style.left = (Math.cos(angle) * 200) + "px";
    requestAnimationFrame(newTime => animate(newTime, time));
  }
  requestAnimationFrame(animate);
</script>
```

La flecha gris muestra la trayectoria a lo largo de la cual se mueve la imagen.



Nuestra imagen está centrada en la página y tiene una posición de `relative`. Actualizaremos repetidamente los estilos `top` e `left` de esa imagen para moverla.

El script utiliza `requestAnimationFrame` para programar la ejecución de la función `animar` siempre que el navegador esté listo para repintar la pantalla. La función `animar` a su vez vuelve a llamar a `requestAnimationFrame` para programar la siguiente actualización. Cuando la ventana del navegador (o pestaña) está activa, esto provocará que las actualizaciones ocurran a una velocidad de aproximadamente 60 por segundo, lo que suele producir una animación atractiva.

Si simplemente actualizáramos el DOM en un bucle, la página se congelaría y nada aparecería en la pantalla. Los navegadores no actualizan su pantalla mientras se ejecuta un programa JavaScript, ni permiten ninguna interacción con la página. Por eso necesitamos `requestAnimationFrame` — le indica al navegador que hemos terminado por ahora, y puede continuar haciendo las cosas que hacen los navegadores, como actualizar la pantalla y responder a las acciones del usuario.

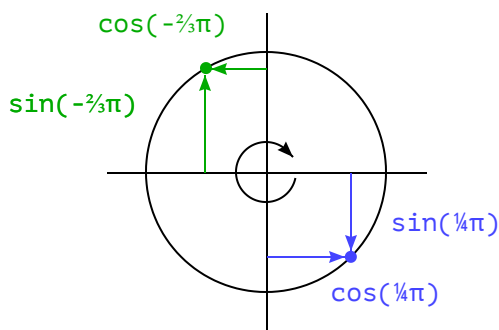
La función de animación recibe el tiempo actual como argumento. Para asegurar que el movimiento del gato por milisegundo sea estable, basa la velocidad a la que cambia el ángulo en la diferencia entre el tiempo actual y el último tiempo en que se ejecutó la función. Si simplemente moviera el ángulo por una cantidad fija por paso, el movimiento se interrumpiría si, por ejemplo, otra tarea pesada que se está ejecutando en la misma computadora impidiera que la función se ejecutara durante una fracción de segundo.

Moverse en círculos se hace utilizando las funciones trigonométricas `Math.cos` y `Math.sin`. Para aquellos que no estén familiarizados

con ellas, las presentaré brevemente ya que ocasionalmente las utilizaremos en este libro.

`Math.cos` y `Math.sin` son útiles para encontrar puntos que se encuentran en un círculo alrededor del punto (0,0) con un radio de uno. Ambas funciones interpretan su argumento como la posición en este círculo, con cero denotando el punto en el extremo derecho del círculo, avanzando en el sentido de las agujas del reloj hasta que 2π (aproximadamente 6,28) nos ha llevado alrededor de todo el círculo. `Math.cos` te indica la coordenada x del punto que corresponde a la posición dada, y `Math.sin` devuelve la coordenada y. Las posiciones (o ángulos) mayores que 2π o menores que 0 son válidos, la rotación se repite de manera que $a+2\pi$ se refiere al mismo ángulo que a .

Esta unidad para medir ángulos se llama radianes — un círculo completo son 2π radianes, similar a cómo son 360 grados al medir en grados. La constante π está disponible como `Math.PI` en JavaScript.



El código de animación del gato mantiene un contador, `angle`, para el ángulo actual de la animación e incrementa el mismo cada vez que se llama la función `animate`. Luego puede usar este ángulo para calcular la posición actual del elemento de imagen. El estilo `top` es

calculado con `Math.sin` y multiplicado por 20, que es el radio vertical de nuestra elipse. El estilo `left` se basa en `Math.cos` y multiplicado por 200 para que la elipse sea mucho más ancha que alta.

Ten en cuenta que los estilos usualmente necesitan *unidades*. En este caso, tenemos que añadir "px" al número para indicarle al navegador que estamos contando en píxeles (en lugar de centímetros, "ems" u otras unidades). Esto es fácil de olvidar. Usar números sin unidades resultará en que tu estilo sea ignorado — a menos que el número sea 0, lo cual siempre significa lo mismo, independientemente de su unidad.

RESUMEN

Los programas de JavaScript pueden inspeccionar e interferir con el documento que el navegador está mostrando a través de una estructura de datos llamada el DOM. Esta estructura de datos representa el modelo del documento del navegador, y un programa de JavaScript puede modificarlo para cambiar el documento visible.

El DOM está organizado como un árbol, en el cual los elementos están dispuestos jerárquicamente de acuerdo a la estructura del documento. Los objetos que representan elementos tienen propiedades como `parentNode` y `childNodes`, las cuales pueden ser usadas para navegar a través de este árbol.

La forma en que un documento es mostrado puede ser influenciada por el *estilo*, tanto adjuntando estilos directamente a nodos como definiendo reglas que coincidan con ciertos nodos. Hay muchas propiedades de estilo diferentes, como `color` o `display`. El código

de JavaScript puede manipular el estilo de un elemento directamente a través de su propiedad `style`.

EJERCICIOS

CONSTRUIR UNA TABLA

Una tabla HTML se construye con la siguiente estructura de etiquetas:

```
<table>
  <tr>
    <th>nombre</th>
    <th>altura</th>
    <th>lugar</th>
  </tr>
  <tr>
    <td>Kilimanjaro</td>
    <td>5895</td>
    <td>Tanzania</td>
  </tr>
</table>
```

Dado un conjunto de datos de montañas, un array de objetos con propiedades `name`, `height`, y `place`, genera la estructura DOM para una tabla que enumera los objetos. Debería haber una columna por clave y una fila por objeto, además de una fila de encabezado con elementos `<th>` en la parte superior, enumerando los nombres de las columnas.

Escribe esto de manera que las columnas se deriven automáticamente de los objetos, tomando los nombres de las propiedades del primer objeto en los datos.

Muestra la tabla resultante en el documento agregándola al elemento que tenga un atributo `id` de "mountains".

Una vez que tengas esto funcionando, alinea a la derecha las celdas que contienen valores numéricos estableciendo su propiedad `style.textAlign` en "right".

ELEMENTOS POR NOMBRE DE ETIQUETA

El método `document.getElementsByTagName` devuelve todos los elementos hijos con un nombre de etiqueta dado. Implementa tu propia versión de esto como una función que tome un nodo y un string (el nombre de la etiqueta) como argumentos y devuelva un array que contenga todos los nodos de elementos descendientes con el nombre de etiqueta dado. Tu función debe recorrer el documento en sí. No puede usar un método como `querySelectorAll` para hacer el trabajo.

Para encontrar el nombre de etiqueta de un elemento, usa su propiedad `nodeName`. Pero ten en cuenta que esto devolverá el nombre de la etiqueta en mayúsculas. Usa los métodos de string `toLowerCase` o `toUpperCase` para compensar esto.

EL SOMBRERO DEL GATO

Extiende la animación del gato definida anteriormente para que tanto el gato como su sombrero (``) orbiten en lados opuestos de la elipse.

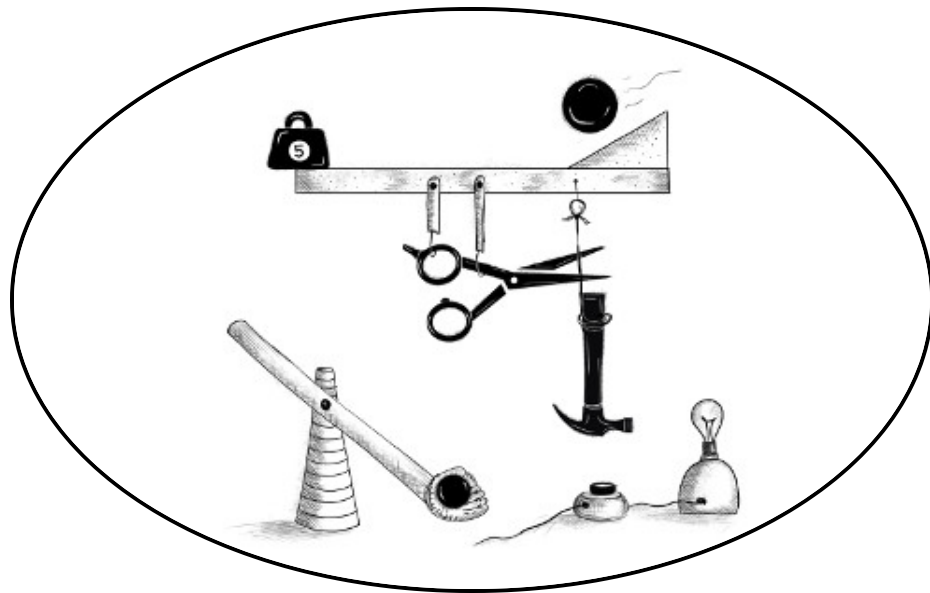
O haz que el sombrero circule alrededor del gato. O altera la animación de alguna otra manera interesante.

Para facilitar el posicionamiento de varios objetos, es probablemente una buena idea cambiar a posicionamiento absoluto. Esto significa que `top` y `left` se cuentan en relación al extremo superior izquierdo del documento. Para evitar usar coordenadas negativas, que harían que la imagen se salga de la página visible, puedes agregar un número fijo de píxeles a los valores de posición.

MANEJO DE EVENTOS

“Tienes poder sobre tu mente, no sobre los eventos externos. Date cuenta de esto y encontrarás fuerza.”

—Marco Aurelio, *Meditaciones*



Algunos programas trabajan con la entrada directa del usuario, como acciones del ratón y del teclado. Ese tipo de entrada no está disponible de antemano, como una estructura de datos bien organizada, llega pieza por pieza, en tiempo real, y el programa debe responder a medida que sucede.

CONTROLADORES DE EVENTOS

Imagina una interfaz donde la única forma de saber si una tecla en el teclado está siendo presionada es leyendo el estado actual de esa tecla. Para poder reaccionar a las pulsaciones de teclas, tendrías que leer constantemente el estado de la tecla para capturarla antes de que se libere nuevamente. Sería peligroso realizar otras computaciones intensivas en tiempo, ya que podrías perder una pulsación de tecla.

Algunas máquinas primitivas manejan la entrada de esa manera. Un paso adelante sería que el hardware o el sistema operativo noten la pulsación de tecla y la pongan en una cola. Un programa puede luego verificar periódicamente la cola en busca de nuevos eventos y reaccionar a lo que encuentre allí.

Por supuesto, tiene que recordar mirar la cola y hacerlo a menudo, porque cualquier tiempo transcurrido entre la presión de la tecla y la notificación del evento por parte del programa hará que el software se sienta sin respuesta. Este enfoque se llama *sondeo*. La mayoría de los programadores prefieren evitarlo.

Un mecanismo mejor es que el sistema notifique activamente a nuestro código cuando ocurre un evento. Los navegadores hacen esto al permitirnos registrar funciones como *manejadores* para eventos específicos.

```
<p>Haz clic en este documento para activar el manejador.</p>
<script>
  window.addEventListener("click", () => {
    console.log("¿Llamaste?");
  });
</script>
```

La asignación `window` se refiere a un objeto integrado proporcionado por el navegador. Representa la ventana del navegador que contiene el documento. Llamar a su método `addEventListener` registra el segundo argumento para que se llame cada vez que ocurra el evento descrito por su primer argumento.

EVENTOS Y NODOS DOM

Cada controlador de eventos del navegador se registra en un contexto. En el ejemplo anterior llamamos a `addEventListener` en el objeto `window` para registrar un controlador para toda la ventana. Un método similar también se encuentra en elementos del DOM y algunos otros tipos de objetos. Los escuchas de eventos solo se llaman cuando el evento ocurre en el contexto del objeto en el que están registrados.

```
<button>Haz clic</button>
<p>No hay manejador aquí.</p>
<script>
  let button = document.querySelector("button");
  button.addEventListener("click", () => {
    console.log("Botón clickeado.");
  });
</script>
```

Ese ejemplo adjunta un manejador al nodo del botón. Los clics en el botón hacen que se ejecute ese manejador, pero los clics en el resto del documento no lo hacen.

Darle a un nodo un atributo `onclick` tiene un efecto similar. Esto funciona para la mayoría de tipos de eventos: puedes adjuntar un

manejador a través del atributo cuyo nombre es el nombre del evento con `on` al inicio.

Pero un nodo solo puede tener un atributo `onclick`, por lo que solo puedes registrar un manejador por nodo de esa manera. El método `addEventListener` te permite agregar cualquier cantidad de manejadores, por lo que es seguro agregar manejadores incluso si ya hay otro manejador en el elemento.

El método `removeEventListener`, llamado con argumentos similares a `addEventListener`, remueve un manejador.

```
<button>Botón de acción única</button>
<script>
  let button = document.querySelector("button");
  function unaVez() {
    console.log("¡Hecho!");
    button.removeEventListener("click", unaVez);
  }
  button.addEventListener("click", unaVez);
</script>
```

La función proporcionada a `removeEventListener` debe ser el mismo valor de función que se proporcionó a `addEventListener`. Por lo tanto, para anular el registro de un manejador, querrás darle un nombre a la función (`unaVez`, en el ejemplo) para poder pasar el mismo valor de función a ambos métodos.

OBJETOS DE EVENTOS

Aunque lo hemos ignorado hasta ahora, las funciones de manejadores de eventos reciben un argumento: el *objeto de evento*. Este objeto contiene información adicional sobre el evento. Por

ejemplo, si queremos saber *cuál* botón del mouse se presionó, podemos mirar la propiedad `button` del objeto de evento.

```
<button>Haz clic como quieras</button>
<script>
  let button = document.querySelector("button");
  button.addEventListener("mousedown", event => {
    if (event.button == 0) {
      console.log("Botón izquierdo");
    } else if (event.button == 1) {
      console.log("Botón del medio");
    } else if (event.button == 2) {
      console.log("Botón derecho");
    }
  });
</script>
```

La información almacenada en un objeto de evento difiere según el tipo de evento. Discutiremos diferentes tipos más adelante en el capítulo. La propiedad `type` del objeto siempre contiene una cadena que identifica el evento (como `"click"` o `"mousedown"`).

PROPAGACIÓN

Para la mayoría de tipos de evento, los manejadores registrados en nodos con hijos también recibirán eventos que ocurran en los hijos. Si se hace clic en un botón dentro de un párrafo, los manejadores de eventos en el párrafo también verán el evento de clic.

Pero si tanto el párrafo como el botón tienen un controlador, el controlador más específico —el del botón— tiene prioridad para ejecutarse primero. Se dice que el evento *se propaga* hacia afuera, desde el nodo donde ocurrió hacia el nodo padre de ese nodo y hasta la raíz del documento. Finalmente, después de que todos los

controladores registrados en un nodo específico hayan tenido su turno, los controladores registrados en toda la ventana tienen la oportunidad de responder al evento.

En cualquier momento, un controlador de eventos puede llamar al método `stopPropagation` en el objeto de evento para evitar que los controladores superiores reciban el evento. Esto puede ser útil cuando, por ejemplo, tienes un botón dentro de otro elemento clickeable y no quieres que los clics en el botón activen el comportamiento de click del elemento externo.

El siguiente ejemplo registra controladores de `"mousedown"` tanto en un botón como en el párrafo que lo rodea. Cuando se hace clic con el botón derecho del ratón, el controlador del botón llama a `stopPropagation`, lo que evitará que se ejecute el controlador en el párrafo. Cuando el botón se hace clic con otro botón del ratón, ambos controladores se ejecutarán.

```
<p>Un párrafo con un <button>botón</button>.</p>
<script>
  let para = document.querySelector("p");
  let button = document.querySelector("button");
  para.addEventListener("mousedown", () => {
    console.log("Controlador para el párrafo.");
  });
  button.addEventListener("mousedown", event => {
    console.log("Controlador para el botón.");
    if (event.button == 2) event.stopPropagation();
  });
</script>
```

La mayoría de los objetos de eventos tienen una propiedad `target` que se refiere al nodo donde se originaron. Puedes usar esta propiedad para asegurarte de que no estás manejando

accidentalmente algo que se propagó desde un nodo que no deseas manejar.

También es posible usar la propiedad `target` para abarcar un amplio rango para un tipo específico de evento. Por ejemplo, si tienes un nodo que contiene una larga lista de botones, puede ser más conveniente registrar un único controlador de clic en el nodo externo y hacer que utilice la propiedad `target` para averiguar si se hizo clic en un botón, en lugar de registrar controladores individuales en todos los botones.

```
<button>A</button>
<button>B</button>
<button>C</button>
<script>
  document.body.addEventListener("click", event => {
    if (event.target.nodeName == "BUTTON") {
      console.log("Clic en", event.target.textContent);
    }
  });
</script>
```

ACCIONES PREDETERMINADAS

Muchos eventos tienen una acción predeterminada asociada a ellos. Si haces clic en un enlace, serás llevado al destino del enlace. Si presionas la flecha hacia abajo, el navegador desplazará la página hacia abajo. Si haces clic derecho, obtendrás un menú contextual. Y así sucesivamente.

Para la mayoría de los tipos de eventos, los controladores de eventos de JavaScript se ejecutan *antes* de que ocurra el comportamiento predeterminado. Si el controlador no desea que este comportamiento

normal ocurra, típicamente porque ya se encargó de manejar el evento, puede llamar al método `preventDefault` en el objeto de evento.

Esto se puede utilizar para implementar tus propios atajos de teclado o menús contextuales. También se puede usar para interferir de manera molesta con el comportamiento que los usuarios esperan.

Por ejemplo, aquí hay un enlace que no se puede seguir:

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  let link = document.querySelector("a");
  link.addEventListener("click", event => {
    console.log("¡Incorrecto!");
    event.preventDefault();
  });
</script>
```

Trata de no hacer este tipo de cosas a menos que tengas una razón realmente válida. Será desagradable para las personas que utilicen tu página cuando se rompa el comportamiento esperado.

Dependiendo del navegador, algunos eventos no se pueden interceptar en absoluto. En Chrome, por ejemplo, el atajo de teclado para cerrar la pestaña actual (control-W o command-W) no se puede manejar con JavaScript.

EVENTOS DE TECLADO

Cuando se presiona una tecla en el teclado, tu navegador dispara un evento `"keydown"`. Cuando se suelta, obtienes un evento `"keyup"`.

```
<p>Esta página se vuelve violeta cuando mantienes presionada la
tecla V.</p>
```

```
<script>
  window.addEventListener("keydown", event => {
    if (event.key == "v") {
      document.body.style.background = "violet";
    }
  });
  window.addEventListener("keyup", event => {
    if (event.key == "v") {
      document.body.style.background = "";
    }
  });
}</script>
```

A pesar de su nombre, "keydown" se dispara no solo cuando la tecla se presiona físicamente hacia abajo. Cuando se presiona y se mantiene una tecla, el evento se vuelve a disparar cada vez que la tecla *se repite*. A veces tienes que tener cuidado con esto. Por ejemplo, si agregas un botón al DOM cuando se presiona una tecla y lo eliminas de nuevo cuando se suelta la tecla, podrías agregar accidentalmente cientos de botones cuando se mantiene presionada la tecla durante más tiempo.

El ejemplo observó la propiedad `key` del objeto evento para ver sobre qué tecla es el evento. Esta propiedad contiene una cadena que, para la mayoría de las teclas, corresponde a lo que escribirías al presionar esa tecla. Para teclas especiales como `ENTER`, contiene una cadena que nombra la tecla ("Enter" , en este caso). Si mantienes presionado `SHIFT` mientras presionas una tecla, eso también puede influir en el nombre de la tecla: "v" se convierte en "V", y "1" puede convertirse en "!", si eso es lo que produce al presionar `SHIFT-1` en tu teclado.

Las teclas modificadoras como `SHIFT`, `CONTROL`, `ALT` y `META` (command en Mac) generan eventos de tecla igual que las teclas normales. Pero

al buscar combinaciones de teclas, también puedes averiguar si estas teclas se mantienen presionadas mirando las propiedades `shiftKey`, `ctrlKey`, `altKey` y `metaKey` de los eventos de teclado y ratón.

```
<p>Pulsa Control-Espacio para continuar.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == " " && event.ctrlKey) {
      console.log("¡Continuando!");
    }
  });
</script>
```

El nodo del DOM donde se origina un evento de teclado depende del elemento que tiene foco cuando se presiona la tecla. La mayoría de los nodos no pueden tener foco a menos que les des un atributo `tabindex`, pero cosas como los enlaces, botones y campos de formulario pueden. Volveremos a los campos de formulario en [Capítulo 18](#). Cuando nada en particular tiene foco, `document.body` actúa como el nodo objetivo de los eventos de teclado.

Cuando el usuario está escribiendo texto, utilizar eventos de teclado para averiguar qué se está escribiendo es problemático. Algunas plataformas, especialmente el teclado virtual en teléfonos Android, no disparan eventos de teclado. Pero incluso cuando se tiene un teclado tradicional, algunos tipos de entrada de texto no coinciden con las pulsaciones de teclas de manera directa, como el software de *editor de método de entrada* (IME) utilizado por personas cuyos guiones no caben en un teclado, donde múltiples pulsaciones de teclas se combinan para crear caracteres.

Para detectar cuando se ha escrito algo, los elementos en los que se puede escribir, como las etiquetas `<input>` y `<textarea>`, activan eventos `"input"` cada vez que el usuario cambia su contenido. Para obtener el contenido real que se ha escrito, lo mejor es leerlo directamente del campo enfocado. [Capítulo 18](#) mostrará cómo hacerlo.

EVENTOS DE PUNTERO

Actualmente existen dos formas ampliamente utilizadas de señalar cosas en una pantalla: los ratones (incluyendo dispositivos que actúan como ratones, como touchpads y trackballs) y las pantallas táctiles. Estas producen diferentes tipos de eventos.

CLICS DE RATÓN

Presionar un botón de ratón provoca que se disparen varios eventos. Los eventos `"mousedown"` y `"mouseup"` son similares a `"keydown"` y `"keyup"` y se activan cuando se presiona y se suelta el botón. Estos eventos ocurren en los nodos del DOM que están inmediatamente debajo del puntero del ratón cuando se produce el evento.

Después del evento `"mouseup"`, se dispara un evento `"click"` en el nodo más específico que contenía tanto la pulsación como la liberación del botón. Por ejemplo, si presiono el botón del ratón en un párrafo y luego muevo el puntero a otro párrafo y suelto el botón, el evento `"click"` ocurrirá en el elemento que contiene ambos párrafos.

Si dos clics ocurren cerca uno del otro, también se dispara un evento "dblclick" (doble clic), después del segundo evento de clic.

Para obtener información precisa sobre el lugar donde ocurrió un evento de ratón, puedes mirar sus propiedades `clientX` y `clientY`, que contienen las coordenadas del evento (en píxeles) relativas a la esquina superior izquierda de la ventana, o `pageX` y `pageY`, que son relativas a la esquina superior izquierda de todo el documento (lo cual puede ser diferente cuando la ventana ha sido desplazada).

El siguiente programa implementa una aplicación de dibujo primitiva. Cada vez que haces clic en el documento, agrega un punto bajo el puntero de tu ratón. Ver [Capítulo 19](#) para una aplicación de dibujo menos primitiva.

```
<style>
  body {
    height: 200px;
    background: beige;
  }
  .dot {
    height: 8px; width: 8px;
    border-radius: 4px; /* redondea las esquinas */
    background: teal;
    position: absolute;
  }
</style>
<script>
  window.addEventListener("click", event => {
    let dot = document.createElement("div");
    dot.className = "dot";
    dot.style.left = (event.pageX - 4) + "px";
    dot.style.top = (event.pageY - 4) + "px";
    document.body.appendChild(dot);
  });
</script>
```

```
});  
</script>
```

MOVIMIENTO DEL RATÓN

Cada vez que el puntero del ratón se mueve, se dispara un evento "mousemove". Este evento se puede usar para rastrear la posición del ratón. Una situación común en la que esto es útil es al implementar algún tipo de funcionalidad de arrastrar y soltar con el ratón.

Como ejemplo, el siguiente programa muestra una barra y configura controladores de eventos para que al arrastrar hacia la izquierda o hacia la derecha en esta barra, se haga más estrecha o más ancha:

```
<p>Arrastra la barra para cambiar su anchura:</p>  
<div style="background: orange; width: 60px; height: 20px">  
</div>  
<script>  
  let lastX; // Rastrea la última posición X del ratón observada  
  let bar = document.querySelector("div");  
  bar.addEventListener("mousedown", event => {  
    if (event.button == 0) {  
      lastX = event.clientX;  
      window.addEventListener("mousemove", moved);  
      event.preventDefault(); // Prevenir selección  
    }  
  });  
  
  function moved(event) {  
    if (event.buttons == 0) {  
      window.removeEventListener("mousemove", moved);  
    } else {  
      let dist = event.clientX - lastX;  
      let newWidth = Math.max(10, bar.offsetWidth + dist);  
      bar.style.width = newWidth + "px";  
      lastX = event.clientX;  
    }  
  }  
</script>
```

```
}  
</script>
```

La página resultante se ve así:

Drag the bar to change its width:



Ten en cuenta que el controlador "mousemove" está registrado en toda la window. Incluso si el ratón sale de la barra durante el cambio de tamaño, mientras el botón se mantenga presionado todavía queremos actualizar su tamaño.

Debemos detener el cambio de tamaño de la barra cuando se libere el botón del ratón. Para eso, podemos usar la propiedad `buttons` (notar el plural), que nos indica qué botones están actualmente presionados. Cuando este valor es cero, ningún botón está presionado. Cuando se mantienen presionados botones, su valor es la suma de los códigos de esos botones—el botón izquierdo tiene el código 1, el derecho 2 y el central 4. Con el botón izquierdo y el derecho presionados, por ejemplo, el valor de `buttons` será 3.

Es importante destacar que el orden de estos códigos es diferente al utilizado por `button`, donde el botón central venía antes que el derecho. Como se mencionó, la consistencia no es realmente un punto fuerte de la interfaz de programación del navegador.

EVENTOS TÁCTILES

El estilo de navegador gráfico que usamos fue diseñado pensando en interfaces de ratón, en una época donde las pantallas táctiles eran raras. Para hacer que la web “funcione” en los primeros teléfonos con pantalla táctil, los navegadores de esos dispositivos fingían, hasta cierto punto, que los eventos táctiles eran eventos de ratón. Si tocas la pantalla, recibirás eventos de `"mousedown"`, `"mouseup"` y `"click"`.

Pero esta ilusión no es muy robusta. Una pantalla táctil funciona de manera diferente a un ratón: no tiene múltiples botones, no se puede rastrear el dedo cuando no está en la pantalla (para simular `"mousemove"`), y permite que varios dedos estén en la pantalla al mismo tiempo.

Los eventos de ratón solo cubren la interacción táctil en casos sencillos: si agregas un controlador de `"click"` a un botón, los usuarios táctiles aún podrán usarlo. Pero algo como la barra redimensionable del ejemplo anterior no funciona en una pantalla táctil.

Existen tipos específicos de eventos disparados por la interacción táctil. Cuando un dedo comienza a tocar la pantalla, se genera un evento `"touchstart"`. Cuando se mueve mientras toca, se generan eventos `"touchmove"`. Finalmente, cuando deja de tocar la pantalla, verás un evento `"touchend"`.

Debido a que muchas pantallas táctiles pueden detectar varios dedos al mismo tiempo, estos eventos no tienen un único conjunto de coordenadas asociadas. Más bien, sus objetos de eventos tienen una propiedad `touches`, que contiene un objeto similar a un array de

puntos, cada uno con sus propias propiedades `clientX`, `clientY`, `pageX` y `pageY`.

Podrías hacer algo como esto para mostrar círculos rojos alrededor de cada dedo que toca:

```
<style>
  dot { position: absolute; display: block;
        border: 2px solid red; border-radius: 50px;
        height: 100px; width: 100px; }
</style>
<p>Toca esta página</p>
<script>
  function update(event) {
    for (let dot; dot = document.querySelector("dot");) {
      dot.remove();
    }
    for (let i = 0; i < event.touches.length; i++) {
      let {pageX, pageY} = event.touches[i];
      let dot = document.createElement("dot");
      dot.style.left = (pageX - 50) + "px";
      dot.style.top = (pageY - 50) + "px";
      document.body.appendChild(dot);
    }
  }
  window.addEventListener("touchstart", update);
  window.addEventListener("touchmove", update);
  window.addEventListener("touchend", update);
</script>
```

A menudo querrás llamar a `preventDefault` en los controladores de eventos táctiles para anular el comportamiento predeterminado del navegador (que puede incluir desplazar la página al deslizar) y evitar que se generen eventos de ratón, para los cuales también puedes tener un controlador.

EVENTOS DE DESPLAZAMIENTO

Cada vez que un elemento se desplaza, se dispara un evento "scroll". Esto tiene varios usos, como saber qué está viendo actualmente el usuario (para desactivar animaciones fuera de la pantalla o enviar informes de vigilancia a tu malvada sede) o mostrar alguna indicación de progreso (resaltando parte de una tabla de contenidos o mostrando un número de página). El siguiente ejemplo dibuja una barra de progreso sobre el documento y la actualiza para llenarla a medida que se desplaza hacia abajo:

```
<style>
  #progress {
    border-bottom: 2px solid blue;
    width: 0;
    position: fixed;
    top: 0; left: 0;
  }
</style>
<div id="progress"></div>
<script>
  // Create some content
  document.body.appendChild(document.createTextNode(
    "supercalifragilisticexpialidocious ".repeat(1000)));

  let bar = document.querySelector("#progress");
  window.addEventListener("scroll", () => {
    let max = document.body.scrollHeight - innerHeight;
    bar.style.width = `${(pageYOffset / max) * 100}%`;
  });
</script>
```

Darle a un elemento una `position` de `fixed` actúa de manera similar a una posición `absolute`, pero también evita que se desplace junto con el resto del documento. El efecto es hacer que nuestra barra de progreso permanezca en la parte superior. Su ancho se cambia para indicar el progreso actual. Usamos `%`, en lugar de `px`,

como unidad al establecer el ancho para que el elemento tenga un tamaño relativo al ancho de la página.

El enlace global `innerHeight` nos da la altura de la ventana, que debemos restar de la altura total desplazable, ya que no se puede seguir desplazando cuando se llega al final del documento. También existe un `innerWidth` para el ancho de la ventana. Al dividir `pageYOffset`, la posición actual de desplazamiento, por la posición máxima de desplazamiento y multiplicar por 100, obtenemos el porcentaje para la barra de progreso.

Llamar a `preventDefault` en un evento de desplazamiento no impide que ocurra el desplazamiento. De hecho, el controlador de eventos se llama solo *después* de que ocurre el desplazamiento.

EVENTOS DE ENFOQUE

Cuando un elemento recibe el enfoque, el navegador dispara un evento `"focus"` en él. Cuando pierde el enfoque, el elemento recibe un evento `"blur"`.

A diferencia de los eventos discutidos anteriormente, estos dos eventos no se propagan. Un controlador en un elemento padre no recibe notificaciones cuando un elemento hijo recibe o pierde el enfoque.

El siguiente ejemplo muestra texto de ayuda para el campo de texto que actualmente tiene el foco:

```
<p>Nombre: <input type="text" data-help="Tu nombre completo"></p>
<p>Edad: <input type="text" data-help="Tu edad en años"></p>
<p id="help"></p>
```

```

<script>
  let help = document.querySelector("#help");
  let fields = document.querySelectorAll("input");
  for (let field of Array.from(fields)) {
    field.addEventListener("focus", event => {
      let text = event.target.getAttribute("data-help");
      help.textContent = text;
    });
    field.addEventListener("blur", event => {
      help.textContent = "";
    });
  }
</script>

```

Esta captura de pantalla muestra el texto de ayuda para el campo de edad.

Name:

Age:

Age in years

El objeto ((window)) recibirá eventos "focus" y "blur" cuando el usuario se mueva desde o hacia la pestaña o ventana del navegador en la que se muestra el documento.

EVENTO DE CARGA

Cuando una página termina de cargarse, se dispara el evento "load" en los objetos ventana y cuerpo del documento. Esto se usa a menudo para programar acciones de inicialización que requieren que todo el documento haya sido construido. Recuerda que el contenido de las etiquetas <script> se ejecuta inmediatamente

cuando se encuentra la etiqueta. Esto puede ser demasiado pronto, por ejemplo, cuando el script necesita hacer algo con partes del documento que aparecen después de la etiqueta `<script>`.

Elementos como imágenes y etiquetas de script que cargan un archivo externo también tienen un evento "load" que indica que se cargaron los archivos a los que hacen referencia. Al igual que los eventos relacionados con el enfoque, los eventos de carga no se propagan.

Cuando se cierra una página o se navega lejos de ella (por ejemplo, al seguir un enlace), se dispara un evento "beforeunload". El uso principal de este evento es evitar que el usuario pierda accidentalmente su trabajo al cerrar un documento. Si previenes el comportamiento predeterminado en este evento *y* estableces la propiedad `returnValue` en el objeto de evento a una cadena, el navegador mostrará al usuario un cuadro de diálogo preguntando si realmente desea abandonar la página. Ese cuadro de diálogo podría incluir tu cadena, pero debido a que algunos sitios maliciosos intentan usar estos cuadros de diálogo para confundir a las personas y hacer que se queden en su página para ver anuncios de pérdida de peso dudosos, la mayoría de los navegadores ya no los muestran.

EVENTOS Y EL BUCLE DE EVENTOS

En el contexto del bucle de eventos, como se discutió en [Capítulo 11](#), los controladores de eventos del navegador se comportan como otras notificaciones asíncronas. Se programan cuando ocurre el evento pero deben esperar a que otros scripts que se estén ejecutando terminen antes de tener la oportunidad de ejecutarse.

El hecho de que los eventos solo se puedan procesar cuando no hay nada más en ejecución significa que, si el bucle de eventos está ocupado con otro trabajo, cualquier interacción con la página (que ocurre a través de eventos) se retrasará hasta que haya tiempo para procesarla. Entonces, si programas demasiado trabajo, ya sea con controladores de eventos de larga duración o con muchos que se ejecutan rápidamente, la página se volverá lenta y pesada de usar.

Para casos en los que *realmente* quieres hacer algo que consume mucho tiempo en segundo plano sin congelar la página, los navegadores proporcionan algo llamado *web workers*. Un worker es un proceso de JavaScript que se ejecuta junto al script principal, en su propia línea de tiempo.

Imagina que elevar al cuadrado un número es una computación pesada y de larga duración que queremos realizar en un hilo separado. Podríamos escribir un archivo llamado `code/squareworker.js` que responda a mensajes calculando un cuadrado y enviando un mensaje de vuelta.

```
addEventListener("message", event => {  
  postMessage(event.data * event.data);  
});
```

Para evitar los problemas de tener múltiples hilos tocando los mismos datos, los workers no comparten su alcance global ni ningún otro dato con el entorno del script principal. En cambio, debes comunicarte con ellos enviando mensajes de ida y vuelta.

Este código genera un worker que ejecuta ese script, le envía algunos mensajes y muestra las respuestas.

```
let squareWorker = new Worker("code/squareworker.js");
squareWorker.addEventListener("message", event => {
  console.log("El worker respondió:", event.data);
});
squareWorker.postMessage(10);
squareWorker.postMessage(24);
```

La función `postMessage` envía un mensaje, lo que causará que se dispare un evento `"message"` en el receptor. El script que creó el worker envía y recibe mensajes a través del objeto `Worker`, mientras que el worker se comunica con el script que lo creó enviando y escuchando directamente en su alcance global. Solo se pueden enviar como mensajes valores que puedan representarse como JSON; el otro lado recibirá una *copia* de ellos en lugar del valor en sí mismo.

TEMPORIZADORES

Vimos la función `setTimeout` en [Capítulo 11](#). Programa otra función para que se llame más tarde, después de un cierto número de milisegundos.

A veces necesitas cancelar una función que has programado. Esto se hace almacenando el valor devuelto por `setTimeout` y llamando a `clearTimeout` sobre él.

```
let bombTimer = setTimeout(() => {
  console.log("¡BOOM!");
}, 500);

if (Math.random() < 0.5) { // 50% de probabilidad
  console.log("Desactivado.");
  clearTimeout(bombTimer);
}
```

La función `cancelAnimationFrame` funciona de la misma manera que `clearTimeout`; llamarla en un valor devuelto por `requestAnimationFrame` cancelará ese fotograma (si no se ha llamado ya).

Un conjunto similar de funciones, `setInterval` y `clearInterval`, se utilizan para programar temporizadores que deben *repetirse* cada *X* milisegundos.

```
let ticks = 0;
let reloj = setInterval(() => {
  console.log("tic", ticks++);
  if (ticks == 10) {
    clearInterval(reloj);
    console.log("¡Detener!");
  }
}, 200);
```

DEBOUNCING

Algunos tipos de eventos pueden activarse rápidamente, muchas veces seguidas (como los eventos `"mousemove"` y `"scroll"`, por ejemplo). Al manejar tales eventos, debes tener cuidado de no hacer nada que consuma demasiado tiempo, ya que tu controlador tomará tanto tiempo que la interacción con el documento comenzará a sentirse lenta.

Si necesitas hacer algo importante en un controlador de este tipo, puedes usar `setTimeout` para asegurarte de que no lo estás haciendo con demasiada frecuencia. Esto suele llamarse *debouncing* el evento. Hay varios enfoques ligeramente diferentes para esto.

En el primer ejemplo, queremos reaccionar cuando el usuario ha escrito algo, pero no queremos hacerlo inmediatamente para cada evento de entrada. Cuando están escribiendo rápidamente, solo queremos esperar hasta que ocurra una pausa. En lugar de realizar inmediatamente una acción en el controlador de eventos, establecemos un tiempo de espera. También limpiamos el tiempo de espera anterior (si existe) para que cuando los eventos ocurran cerca uno del otro (más cerca de nuestro retraso de tiempo de espera), el tiempo de espera del evento anterior se cancele.

```
<textarea>Escribe algo aquí...</textarea>
<script>
  let textarea = document.querySelector("textarea");
  let timeout;
  textarea.addEventListener("input", () => {
    clearTimeout(timeout);
    timeout = setTimeout(() => console.log("¡Escrito!"), 500);
  });
</script>
```

Dar un valor no definido a `clearTimeout` o llamarlo en un tiempo de espera que ya ha pasado no tiene efecto. Por lo tanto, no tenemos que tener cuidado de cuándo llamarlo, y simplemente lo hacemos para cada evento.

Podemos usar un patrón ligeramente diferente si queremos espaciar las respuestas para que estén separadas por al menos una cierta longitud de tiempo, pero queremos activarlas *durante* una serie de eventos, no solo después. Por ejemplo, podríamos querer responder a eventos "mousemove" mostrando las coordenadas actuales del mouse pero solo cada 250 milisegundos.

```
<script>
  let programado = null;
  window.addEventListener("mousemove", event => {
    if (!programado) {
      setTimeout(() => {
        document.body.textContent =
          `Ratón en ${programado.pageX}, ${programado.pageY}`;
        programado = null;
      }, 250);
    }
    programado = event;
  });
</script>
```

RESUMEN

Los controladores de eventos hacen posible detectar y reaccionar a eventos que ocurren en nuestra página web. El método `addEventListener` se utiliza para registrar dicho controlador.

Cada evento tiene un tipo (`"keydown"`, `"focus"`, y así sucesivamente) que lo identifica. La mayoría de los eventos se activan en un elemento DOM específico y luego se *propagan* a los ancestros de ese elemento, lo que permite que los controladores asociados a esos elementos los manejen.

Cuando se llama a un controlador de eventos, se le pasa un objeto de evento con información adicional sobre el evento. Este objeto también tiene métodos que nos permiten detener una mayor propagación (`stopPropagation`) y evitar el manejo predeterminado del evento por parte del navegador (`preventDefault`).

Presionar una tecla dispara eventos `"keydown"` y `"keyup"`. Presionar un botón del mouse dispara eventos `"mousedown"`, `"mouseup"` y `"click"`. Mover el mouse dispara eventos `"mousemove"`. La interacción con pantallas táctiles dará lugar a eventos `"touchstart"`, `"touchmove"` y `"touchend"`.

El desplazamiento se puede detectar con el evento `"scroll"`, y los cambios de enfoque se pueden detectar con los eventos `"focus"` y `"blur"`. Cuando el documento ha terminado de cargarse, se activa un evento `"load"` en la ventana.

EJERCICIOS

GLOBO

Escribe una página que muestre un globo (usando el emoji de globo, 🎈). Cuando presiones la flecha hacia arriba, debería inflarse (crecer) un 10 por ciento, y cuando presiones la flecha hacia abajo, debería desinflarse (encoger) un 10 por ciento.

Puedes controlar el tamaño del texto (los emoji son texto) estableciendo la propiedad CSS `font-size` (`style.fontSize`) en su elemento padre. Recuerda incluir una unidad en el valor, por ejemplo, píxeles (`10px`).

Los nombres de las teclas de flecha son `"ArrowUp"` y `"ArrowDown"`. Asegúrate de que las teclas cambien solo el globo, sin hacer scroll en la página.

Cuando eso funcione, añade una característica en la que, si inflas el globo más allá de un cierto tamaño, explote. En este caso, explotar

significa que se reemplace con un emoji de 🌟, y el manejador de eventos se elimine (para que no se pueda inflar o desinflar la explosión).

ESTELA DEL RATÓN

En los primeros días de JavaScript, que fue la época dorada de las páginas de inicio estridentes con un montón de imágenes animadas, la gente ideó formas verdaderamente inspiradoras de usar el lenguaje.

Una de estas era la *estela del ratón* —una serie de elementos que seguirían al puntero del ratón mientras lo movías por la página.

En este ejercicio, quiero que implementes una estela del ratón. Utiliza elementos `<div>` con posición absoluta y un tamaño fijo y color de fondo (consulta el [código](#) en la sección de “Clicks de ratón” para un ejemplo). Crea un montón de estos elementos y, al mover el ratón, muéstralos en la estela del puntero del ratón.

Hay varias aproximaciones posibles aquí. Puedes hacer tu solución tan simple o tan compleja como desees. Una solución simple para empezar es mantener un número fijo de elementos de estela y recorrerlos, moviendo el siguiente a la posición actual del ratón cada vez que ocurra un evento `"mousemove"`.

PESTAÑAS

Los paneles con pestañas son ampliamente utilizados en interfaces de usuario. Te permiten seleccionar un panel de interfaz eligiendo entre varias pestañas que sobresalen por encima de un elemento.

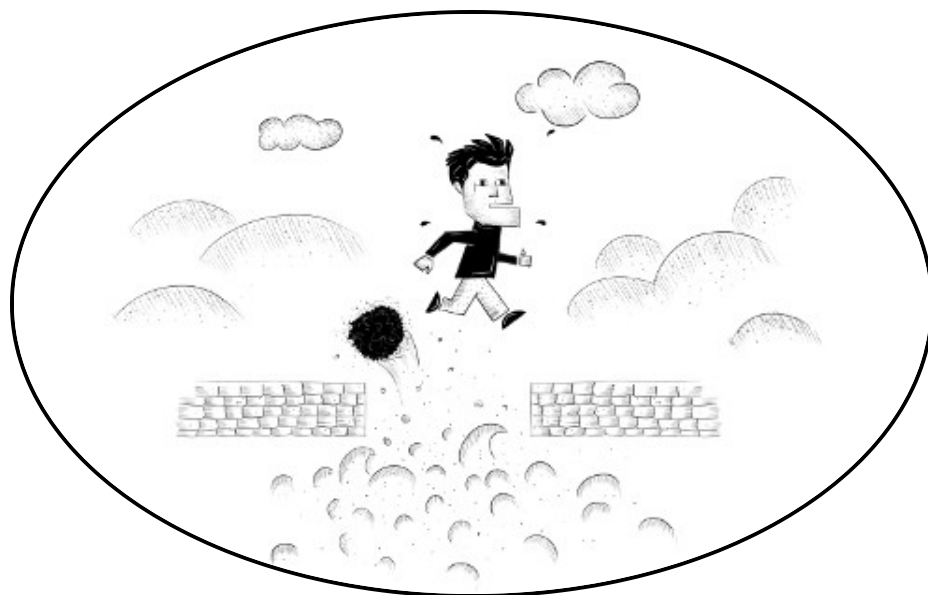
En este ejercicio debes implementar una interfaz de pestañas simple. Escribe una función, `asTabs`, que tome un nodo DOM y cree una interfaz de pestañas que muestre los elementos secundarios de ese nodo. Debería insertar una lista de elementos `<button>` en la parte superior del nodo, uno por cada elemento secundario, conteniendo el texto recuperado del atributo `data-tabname` del hijo. Todos los hijos originales excepto uno deben estar ocultos (con un estilo `display` de `none`). El nodo actualmente visible se puede seleccionar haciendo clic en los botones.

Cuando funcione, extiéndelo para dar estilo al botón de la pestaña actualmente seleccionada de manera diferente para que sea obvio cuál pestaña está seleccionada.

PROYECTO: UN JUEGO DE PLATAFORMAS

“Toda la realidad es un juego.”

—Iain Banks, *The Player of Games*



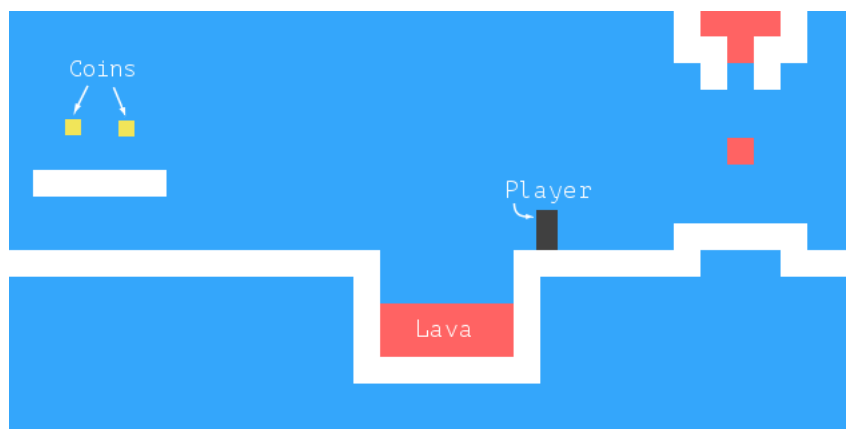
Gran parte de mi fascinación inicial con las computadoras, al igual que la de muchos niños nerds, tenía que ver con los juegos de computadora. Me sentía atraído por los diminutos mundos simulados que podía manipular y en los que se desarrollaban historias (más o menos), supongo, debido a la forma en que proyectaba mi imaginación en ellos más que por las posibilidades que realmente ofrecían.

No le desearía a nadie una carrera en programación de juegos. Al igual que la industria de la música, la discrepancia entre la cantidad de jóvenes entusiastas que desean trabajar en ella y la demanda real de tales personas crea un entorno bastante insalubre. Pero escribir juegos por diversión resulta entretenido.

Este capítulo guiará a través de la implementación de un pequeño juego de plataformas. Los juegos de plataformas (o juegos de “saltos y carreras”) son juegos que esperan que el jugador mueva una figura a través de un mundo, que generalmente es bidimensional y se ve desde el lado, mientras salta sobre y sobre cosas.

EL JUEGO

Nuestro juego estará basado aproximadamente en [Dark Blue](http://www.lessmilk.com/games/10) (www.lessmilk.com/games/10) de Thomas Palef. Elegí ese juego porque es entretenido, minimalista y se puede construir sin mucho código. Se ve así:



La caja oscura representa al jugador, cuya tarea es recolectar las cajas amarillas (monedas) evitando las cosas rojas (lava). Un nivel se completa cuando se han recolectado todas las monedas.

El jugador puede moverse con las teclas de flecha izquierda y derecha y puede saltar con la tecla de flecha hacia arriba. Saltar es una especialidad de este personaje del juego. Puede alcanzar varias veces su altura y puede cambiar de dirección en el aire. Esto puede no ser del todo realista, pero ayuda a darle al jugador la sensación de tener un control directo sobre el avatar en pantalla.

El juego consiste en un fondo estático, dispuesto como una rejilla, con los elementos móviles superpuestos en ese fondo. Cada campo en la rejilla está vacío, sólido o es lava. Los elementos móviles son el jugador, las monedas y ciertas piezas de lava. Las posiciones de estos elementos no están restringidas a la rejilla: sus coordenadas pueden ser fraccionarias, permitiendo un movimiento suave.

LA TECNOLOGÍA

Usaremos el DOM del navegador para mostrar el juego y leeremos la entrada del usuario manejando eventos de teclado.

El código relacionado con la pantalla y el teclado es solo una pequeña parte del trabajo que necesitamos hacer para construir este juego.

Dado que todo se ve como cajas de colores, dibujar es sencillo: creamos elementos del DOM y usamos estilos para darles un color de fondo, tamaño y posición.

Podemos representar el fondo como una tabla ya que es una cuadrícula inmutable de cuadrados. Los elementos de movimiento libre se pueden superponer utilizando elementos posicionados absolutamente.

En juegos y otros programas que deben animar gráficos y responder a la entrada del usuario sin retrasos notables, la eficiencia es importante. Aunque el DOM no fue diseñado originalmente para gráficos de alto rendimiento, en realidad es mejor en esto de lo que podrías esperar. Viste algunas animaciones en [Capítulo 14](#). En una máquina moderna, un juego simple como este funciona bien, incluso si no nos preocupamos mucho por la optimización.

En el [próximo capítulo](#), exploraremos otra tecnología del navegador, la etiqueta `<canvas>`, que proporciona una forma más tradicional de dibujar gráficos, trabajando en términos de formas y píxeles en lugar de elementos del DOM.

NIVELES

Queremos una forma legible y editable por humanos para especificar niveles. Dado que está bien que todo comience en una cuadrícula, podríamos usar cadenas grandes en las que cada carácter represente un elemento, ya sea una parte de la cuadrícula de fondo o un elemento móvil.

El plan para un nivel pequeño podría verse así:

```
let simpleLevelPlan = `
.....
..#.....#..
..#.....=#..
..#.....o.o...#..
..#.@.....#####...#..
..#####.....#..
.....#+++++++#+#..
.....#####..
.....`;
```

Los puntos representan un espacio vacío, los caracteres de almohadilla (#) son paredes y los signos más son lava. La posición inicial del jugador es el signo de arroba (@). Cada carácter O es una moneda, y el signo igual (=) en la parte superior es un bloque de lava que se mueve de un lado a otro horizontalmente.

Además de las dos formas adicionales de lava en movimiento, el carácter de tubería (|) crea blobs que se mueven verticalmente, y v indica lava goteante: lava que se mueve verticalmente y no rebota de un lado a otro, solo se mueve hacia abajo, volviendo a su posición de inicio cuando golpea el suelo.

Un juego completo consta de varios niveles que el jugador debe completar. Un nivel se completa cuando se han recolectado todas las monedas. Si el jugador toca la lava, el nivel actual se restablece a su posición inicial y el jugador puede intentarlo de nuevo.

LEYENDO UN NIVEL

La siguiente clase almacena un objeto nivel. Su argumento debe ser la cadena que define el nivel.

```
class Level {
  constructor(plan) {
    let rows = plan.trim().split("\n").map(l => [...l]);
    this.height = rows.length;
    this.width = rows[0].length;
    this.startActors = [];

    this.rows = rows.map((row, y) => {
      return row.map((ch, x) => {
        let type = levelChars[ch];
        if (typeof type != "string") {
          let pos = new Vec(x, y);
```

```

        this.startActors.push(type.create(pos, ch));
        type = "empty";
    }
    return type;
});
});
}
}

```

El método `trim` se utiliza para eliminar los espacios en blanco al principio y al final de la cadena de plan. Esto permite que nuestro plan de ejemplo comience con una nueva línea para que todas las líneas estén directamente debajo unas de otras. La cadena restante se divide en líneas en caracteres de nueva línea, y cada línea se convierte en un array, produciendo arrays de caracteres.

Entonces, `rows` contiene un array de arrays de caracteres, las filas del plan. Podemos derivar el ancho y alto del nivel a partir de estos. Pero aún debemos separar los elementos móviles de la cuadrícula de fondo. Llamaremos a los elementos móviles *actores*. Se almacenarán en un array de objetos. El fondo será un array de arrays de cadenas, que contienen tipos de campo como "empty", "wall", o "lava".

Para crear estos arrays, mapeamos sobre las filas y luego sobre su contenido. Recuerda que `map` pasa el índice del array como segundo argumento a la función de mapeo, lo que nos indica las coordenadas `x` e `y` de un carácter dado. Las posiciones en el juego se almacenarán como pares de coordenadas, siendo la esquina superior izquierda 0,0 y cada cuadro de fondo siendo de 1 unidad de alto y ancho.

Para interpretar los caracteres en el plan, el constructor de `Level` utiliza el objeto `levelChars`, que, para cada carácter utilizado en las descripciones de niveles, contiene una cadena si es un tipo de

fondo, y una clase si produce un actor. Cuando `type` es una clase de actor, se utiliza su método estático `create` para crear un objeto, que se agrega a `startActors`, y la función de mapeo devuelve `"empty"` para este cuadro de fondo.

La posición del actor se almacena como un objeto `Vec`. Este es un vector bidimensional, un objeto con propiedades `x` e `y`, como se ve en los ejercicios del [Capítulo 6](#).

A medida que el juego avanza, los actores terminarán en lugares diferentes o incluso desaparecerán por completo (como hacen las monedas cuando se recogen). Utilizaremos una clase `State` para seguir el estado de un juego en ejecución.

```
class State {
  constructor(level, actors, status) {
    this.level = level;
    this.actors = actors;
    this.status = status;
  }

  static start(level) {
    return new State(level, level.startActors, "playing");
  }

  get player() {
    return this.actors.find(a => a.type == "player");
  }
}
```

La propiedad `status` cambiará a `"lost"` o `"won"` cuando el juego haya terminado.

Este es nuevamente una estructura de datos persistente: actualizar el estado del juego crea un nuevo estado y deja intacto el anterior.

ACTORES

Los objetos de actores representan la posición actual y el estado de un elemento móvil dado en nuestro juego. Todos los objetos de actores se ajustan a la misma interfaz. Tienen las propiedades `size` y `pos` que contienen el tamaño y las coordenadas de la esquina superior izquierda del rectángulo que representa a este actor.

Luego tienen un método `update`, que se utiliza para calcular su nuevo estado y posición después de un paso de tiempo dado. Simula la acción que realiza el actor: moverse en respuesta a las teclas de flecha para el jugador y rebotar de un lado a otro para la lava, y devuelve un nuevo objeto de actor actualizado.

Una propiedad `type` contiene una cadena que identifica el tipo de actor: `"player"`, `"coin"` o `"lava"`. Esto es útil al dibujar el juego: la apariencia del rectángulo dibujado para un actor se basa en su tipo.

Las clases de actores tienen un método estático `create` que es utilizado por el constructor `Level` para crear un actor a partir de un carácter en el plan de nivel. Recibe las coordenadas del carácter y el carácter en sí, que es necesario porque la clase `Lava` maneja varios caracteres diferentes.

Esta es la clase `Vec` que usaremos para nuestros valores bidimensionales, como la posición y tamaño de los actores.

```
class Vec {
    constructor(x, y) {
        this.x = x; this.y = y;
    }
}
```

```

    plus(other) {
        return new Vec(this.x + other.x, this.y + other.y);
    }
    times(factor) {
        return new Vec(this.x * factor, this.y * factor);
    }
}

```

El método `times` escala un vector por un número dado. Será útil cuando necesitemos multiplicar un vector de velocidad por un intervalo de tiempo para obtener la distancia recorrida durante ese tiempo.

Los diferentes tipos de actores tienen sus propias clases debido a que su comportamiento es muy diferente. Definamos estas clases. Llegaremos a sus métodos `update` más adelante.

La clase `Player` tiene una propiedad `speed` que almacena su velocidad actual para simular el impulso y la gravedad.

```

class Player {
    constructor(pos, speed) {
        this.pos = pos;
        this.speed = speed;
    }

    get type() { return "player"; }

    static create(pos) {
        return new Player(pos.plus(new Vec(0, -0.5)),
                           new Vec(0, 0));
    }
}

Player.prototype.size = new Vec(0.8, 1.5);

```

Dado que un jugador tiene una altura de un cuadro y medio, su posición inicial se establece medio cuadro por encima de la posición donde apareció el carácter @. De esta manera, su parte inferior se alinea con la parte inferior del cuadro en el que apareció.

La propiedad `size` es la misma para todas las instancias de `Player`, por lo que la almacenamos en el prototipo en lugar de en las propias instancias. Podríamos haber utilizado un getter como `type`, pero eso crearía y devolvería un nuevo objeto `Vec` cada vez que se lee la propiedad, lo cual sería derrochador. (Las cadenas, al ser inmutables, no tienen que ser recreadas cada vez que se evalúan).

Al construir un actor `Lava`, necesitamos inicializar el objeto de manera diferente dependiendo del personaje en el que se base. La lava dinámica se mueve a lo largo de su velocidad actual hasta que choca con un obstáculo. En ese momento, si tiene una propiedad de `reset`, saltará de nuevo a su posición de inicio (goteando). Si no la tiene, invertirá su velocidad y continuará en la otra dirección (rebotando).

El método `create` mira el carácter que pasa el constructor de `Level` y crea el actor de lava apropiado.

```
class Lava {
    constructor(pos, speed, reset) {
        this.pos = pos;
        this.speed = speed;
        this.reset = reset;
    }

    get type() { return "lava"; }

    static create(pos, ch) {
        if (ch == "=") {
```

```

        return new Lava(pos, new Vec(2, 0));
    } else if (ch == "|") {
        return new Lava(pos, new Vec(0, 2));
    } else if (ch == "v") {
        return new Lava(pos, new Vec(0, 3), pos);
    }
}
}

```

```
Lava.prototype.size = new Vec(1, 1);
```

Los actores `Coin` son relativamente simples. Mayoritariamente solo se quedan en su lugar. Pero para animar un poco el juego, se les da un “balanceo”, un ligero movimiento vertical de ida y vuelta. Para hacer un seguimiento de esto, un objeto moneda almacena una posición base y también una propiedad de `wobble` que sigue la fase del movimiento de balanceo. Juntos, estos determinan la posición real de la moneda (almacenada en la propiedad `pos`).

```

class Coin {
  constructor(pos, basePos, wobble) {
    this.pos = pos;
    this.basePos = basePos;
    this.wobble = wobble;
  }

  get type() { return "coin"; }

  static create(pos) {
    let basePos = pos.plus(new Vec(0.2, 0.1));
    return new Coin(basePos, basePos,
                    Math.random() * Math.PI * 2);
  }
}

```

```
Coin.prototype.size = new Vec(0.6, 0.6);
```


En [Capítulo 14](#), vimos que `Math.sin` nos da la coordenada y de un punto en un círculo. Esa coordenada va de ida y vuelta en una forma de onda suave a medida que nos movemos a lo largo del círculo, lo que hace que la función seno sea útil para modelar un movimiento ondulado.

Para evitar una situación en la que todas las monedas se mueven hacia arriba y hacia abajo sincrónicamente, la fase inicial de cada moneda se aleatoriza. El periodo de la onda de `Math.sin`, el ancho de una onda que produce, es 2π . Multiplicamos el valor devuelto por `Math.random` por ese número para darle a la moneda una posición inicial aleatoria en la onda.

Ahora podemos definir el objeto `levelChars` que mapea caracteres del plano a tipos de cuadrícula de fondo o clases de actor.

```
const levelChars = {  
  ".": "empty", "#": "wall", "+": "lava",  
  "@": Player, "o": Coin,  
  "=": Lava, "|": Lava, "v": Lava  
};
```

Esto nos brinda todas las partes necesarias para crear una instancia de `Level`.

```
let simpleLevel = new Level(simpleLevelPlan);  
console.log(`${simpleLevel.width} by ${simpleLevel.height}`);  
// → 22 by 9
```

La tarea por delante es mostrar esos niveles en pantalla y modelar el tiempo y movimiento dentro de ellos.

DIBUJO

En el [próximo capítulo](#), mostraremos el mismo juego de una manera diferente. Para hacerlo posible, colocamos la lógica de dibujo detrás de una interfaz y la pasamos al juego como argumento. De esta manera, podemos usar el mismo programa de juego con diferentes nuevos módulos de visualización.

Un objeto de visualización de juego dibuja un nivel y estado dados. Pasamos su constructor al juego para permitir que sea reemplazado. La clase de visualización que definimos en este capítulo se llama `DOMDisplay` porque utiliza elementos del DOM para mostrar el nivel.

Utilizaremos una hoja de estilo para establecer los colores reales y otras propiedades fijas de los elementos que conforman el juego. También sería posible asignar directamente a la propiedad `style` de los elementos al crearlos, pero eso produciría programas más verbosos.

La siguiente función auxiliar proporciona una forma concisa de crear un elemento y darle algunos atributos y nodos secundarios:

```
function elt(nombre, attrs, ...children) {
  let dom = document.createElement(nombre);
  for (let attr of Object.keys(attrs)) {
    dom.setAttribute(attr, attrs[attr]);
  }
  for (let child of children) {
    dom.appendChild(child);
  }
  return dom;
}
```

Una visualización se crea dándole un elemento padre al que debe adjuntarse y un objeto de nivel.

```

class DOMDisplay {
  constructor(padre, nivel) {
    this.dom = elt("div", {class: "game"}, dibujarGrid(nivel));
    this.actorLayer = null;
    padre.appendChild(this.dom);
  }

  clear() { this.dom.remove(); }
}

```

La cuadrícula de fondo del nivel, que nunca cambia, se dibuja una vez. Los actores se vuelven a dibujar cada vez que se actualiza la visualización con un estado dado. La propiedad `actorLayer` se utilizará para realizar un seguimiento del elemento que contiene a los actores para que puedan ser fácilmente eliminados y reemplazados.

Nuestras coordenadas y tamaños se rastrean en unidades de cuadrícula, donde un tamaño o distancia de 1 significa un bloque de cuadrícula. Al establecer tamaños de píxeles, tendremos que escalar estas coordenadas: todo en el juego sería ridículamente pequeño con un solo píxel por cuadrado. La constante `scale` indica el número de píxeles que una unidad ocupa en la pantalla.

```

const escala = 20;

function dibujarGrid(nivel) {
  return elt("table", {
    class: "background",
    style: `width: ${nivel.width * escala}px`
  }, ...nivel.rows.map(fila =>
    elt("tr", {style: `height: ${escala}px`},
      ...fila.map(tipo => elt("td", {class: tipo})))
  ));
}

```

El elemento `<table>` se corresponde bien con la estructura de la propiedad `rows` del nivel: cada fila de la cuadrícula se convierte en una fila de tabla (`<tr>`). Las cadenas en la cuadrícula se usan como nombres de clase para los elementos de celda de tabla (`<td>`). El código utiliza el operador de propagación (triple punto) para pasar matrices de nodos secundarios a `elt` como argumentos separados. El siguiente CSS hace que la tabla se vea como el fondo que queremos:

```
.background    { background: rgb(52, 166, 251);  
                  table-layout: fixed;  
                  border-spacing: 0;                }  
.background td { padding: 0;                        }  
.lava          { background: rgb(255, 100, 100); }  
.wall          { background: white;                }
```

Algunos de estos (`table-layout`, `border-spacing` y `padding`) se utilizan para suprimir comportamientos predeterminados no deseados. No queremos que el diseño de la tabla dependa del contenido de sus celdas, ni queremos espacio entre las celdas de la tabla o relleno dentro de ellas.

La regla `background` establece el color de fondo. CSS permite que los colores se especifiquen tanto como palabras (`white`) como con un formato como `rgb(R, G, B)`, donde los componentes rojo, verde y azul del color se separan en tres números de 0 a 255. Por lo tanto, en `rgb(52, 166, 251)`, el componente rojo es 52, el verde es 166 y el azul es 251. Dado que el componente azul es el más grande, el color resultante será azulado. En la regla `.lava`, el primer número (rojo) es el más grande.

Dibujamos cada actor creando un elemento DOM para él y estableciendo la posición y el tamaño de ese elemento en función de las propiedades del actor. Los valores tienen que ser multiplicados por `scale` para pasar de unidades de juego a píxeles.

```
function drawActors(actors) {  
  return elt("div", {}, ...actors.map(actor => {  
    let rect = elt("div", {class: `actor ${actor.type}`});  
    rect.style.width = `${actor.size.x * scale}px`;  
    rect.style.height = `${actor.size.y * scale}px`;  
    rect.style.left = `${actor.pos.x * scale}px`;  
    rect.style.top = `${actor.pos.y * scale}px`;  
    return rect;  
  }));  
}
```

Para agregar más de una clase a un elemento, separamos los nombres de las clases por espacios. En el siguiente código CSS mostrado a continuación, la clase `actor` da a los actores su posición absoluta. El nombre de su tipo se utiliza como una clase adicional para darles un color. No tenemos que definir la clase `lava` de nuevo porque estamos reutilizando la clase para las casillas de lava de la cuadrícula que definimos anteriormente.

```
.actor { position: absolute; }  
.coin { background: rgb(241, 229, 89); }  
.player { background: rgb(64, 64, 64); }
```

El método `syncState` se utiliza para que la pantalla muestre un estado dado. Primero elimina los gráficos de actores antiguos, si los hay, y luego vuelve a dibujar los actores en sus nuevas posiciones. Puede ser tentador intentar reutilizar los elementos DOM para actores, pero para que eso funcione, necesitaríamos mucho más trabajo adicional para asociar actores con elementos DOM y

asegurarnos de que eliminamos elementos cuando sus actores desaparecen. Dado que típicamente habrá solo un puñado de actores en el juego, volver a dibujar todos ellos no es costoso.

```
DOMDisplay.prototype.syncState = function(state) {  
  if (this.actorLayer) this.actorLayer.remove();  
  this.actorLayer = drawActors(state.actors);  
  this.dom.appendChild(this.actorLayer);  
  this.dom.className = `game ${state.status}`;  
  this.scrollPlayerIntoView(state);  
};
```

Al agregar el estado actual del nivel como nombre de clase al contenedor, podemos estilizar ligeramente al actor del jugador cuando el juego se gana o se pierde, añadiendo una regla CSS que tenga efecto solo cuando el jugador tiene un elemento ancestro con una clase específica.

```
.lost .player {  
  background: rgb(160, 64, 64);  
}  
.won .player {  
  box-shadow: -4px -7px 8px white, 4px -7px 8px white;  
}
```

Después de tocar la lava, el color del jugador se vuelve rojo oscuro, sugiriendo quemaduras. Cuando se ha recolectado la última moneda, agregamos dos sombras blancas difuminadas, una en la parte superior izquierda y otra en la parte superior derecha, para crear un efecto de halo blanco.

No podemos asumir que el nivel siempre encaja en el *viewport* – el elemento en el que dibujamos el juego. Por eso es necesaria la llamada a `scrollPlayerIntoView`. Se asegura de que si el nivel

sobresale del viewport, desplazemos ese viewport para asegurar que el jugador esté cerca de su centro. El siguiente CSS le da al elemento DOM contenedor del juego un tamaño máximo y asegura que cualquier cosa que sobresalga de la caja del elemento no sea visible. También le damos una posición relativa para que los actores dentro de él estén posicionados de manera relativa a la esquina superior izquierda del nivel.

```
.game {  
  overflow: hidden;  
  max-width: 600px;  
  max-height: 450px;  
  position: relative;  
}
```

En el método `scrollPlayerIntoView`, encontramos la posición del jugador y actualizamos la posición de desplazamiento del elemento contenedor. Cambiamos la posición de desplazamiento manipulando las propiedades `scrollLeft` y `scrollTop` de ese elemento cuando el jugador está demasiado cerca del borde.

```
DOMDisplay.prototype.scrollPlayerIntoView = function(state) {  
  let width = this.dom.clientWidth;  
  let height = this.dom.clientHeight;  
  let margin = width / 3;  
  
  // El viewport  
  let left = this.dom.scrollLeft, right = left + width;  
  let top = this.dom.scrollTop, bottom = top + height;  
  
  let player = state.player;  
  let center = player.pos.plus(player.size.times(0.5))  
    .times(scale);  
  
  if (center.x < left + margin) {  
    this.dom.scrollLeft = center.x - margin;  
  }
```

```

    } else if (center.x > right - margin) {
        this.dom.scrollLeft = center.x + margin - width;
    }
    if (center.y < top + margin) {
        this.dom.scrollTop = center.y - margin;
    } else if (center.y > bottom - margin) {
        this.dom.scrollTop = center.y + margin - height;
    }
};

```

La forma en que se encuentra el centro del jugador muestra cómo los métodos en nuestro tipo `Vec` permiten que los cálculos con objetos se escriban de una manera relativamente legible. Para encontrar el centro del actor, sumamos su posición (esquina superior izquierda) y la mitad de su tamaño. Ese es el centro en coordenadas de nivel, pero lo necesitamos en coordenadas de píxeles, así que luego multiplicamos el vector resultante por nuestra escala de visualización.

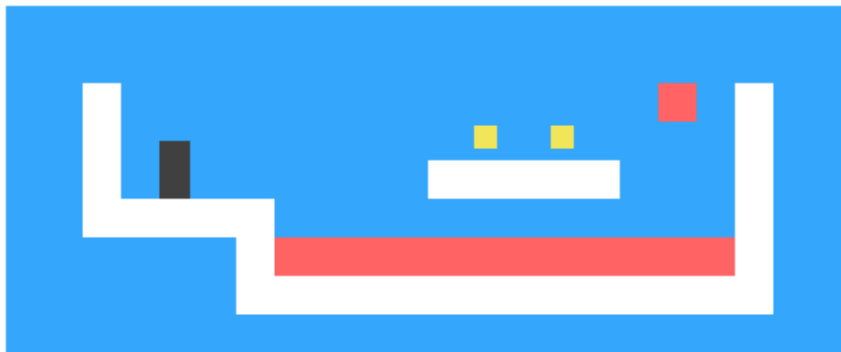
A continuación, una serie de comprobaciones verifica que la posición del jugador no esté fuera del rango permitido. Ten en cuenta que a veces esto establecerá coordenadas de desplazamiento sin sentido que están por debajo de cero o más allá del área desplazable del elemento. Esto está bien, el DOM las limitará a valores aceptables. Establecer `scrollLeft` en -10 hará que se convierta en 0.

Hubiera sido un poco más sencillo intentar siempre desplazar al jugador al centro del viewport. Pero esto crea un efecto bastante brusco. Mientras saltas, la vista se desplazará constantemente hacia arriba y hacia abajo. Es más agradable tener un área “neutral” en el centro de la pantalla donde puedas moverte sin causar ningún desplazamiento.

Ahora podemos mostrar nuestro pequeño nivel.

```
<link rel="stylesheet" href="css/game.css">

<script>
  let simpleLevel = new Level(simpleLevelPlan);
  let display = new DOMDisplay(document.body, simpleLevel);
  display.syncState(State.start(simpleLevel));
</script>
```



La etiqueta `<link>`, cuando se utiliza con `rel="stylesheet"`, es una forma de cargar un archivo CSS en una página. El archivo `game.css` contiene los estilos necesarios para nuestro juego.

MOVIMIENTO Y COLISIÓN

Ahora estamos en el punto en el que podemos comenzar a agregar movimiento. El enfoque básico, seguido por la mayoría de juegos como este, es dividir tiempo en pequeños pasos y, para cada paso, mover a los actores una distancia correspondiente a su velocidad multiplicada por el tamaño del paso de tiempo. Mediremos el tiempo en segundos, por lo que las velocidades se expresan en unidades por segundo.

Mover cosas es fácil. La parte difícil es lidiar con las interacciones entre los elementos. Cuando el jugador golpea una pared o el suelo, no debería simplemente atravesarlo. El juego debe notar cuando un movimiento dado hace que un objeto golpee a otro objeto y responder en consecuencia. Para las paredes, el movimiento debe detenerse. Al golpear una moneda, esa moneda debe ser recogida. Al tocar lava, el juego debería perderse.

Resolver esto para el caso general es una tarea grande. Puedes encontrar bibliotecas, generalmente llamadas *motores físicos*, que simulan la interacción entre objetos físicos en dos o tres dimensiones. Tomaremos un enfoque más modesto en este capítulo, manejando solo colisiones entre objetos rectangulares y manejándolas de una manera bastante simplista.

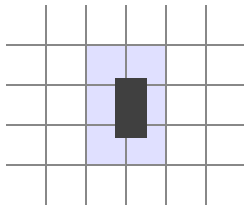
Antes de mover al jugador o un bloque de lava, probamos si el movimiento los llevaría dentro de una pared. Si lo hace, simplemente cancelamos el movimiento por completo. La respuesta a tal colisión depende del tipo de actor. El jugador se detendrá, mientras que un bloque de lava rebotará.

Este enfoque requiere que nuestros pasos de tiempo sean bastante pequeños, ya que hará que el movimiento se detenga antes de que los objetos realmente se toquen. Si los pasos de tiempo (y por lo tanto los pasos de movimiento) son demasiado grandes, el jugador terminaría elevándose a una distancia notable sobre el suelo. Otro enfoque, argumentablemente mejor pero más complicado, sería encontrar el punto exacto de colisión y moverse allí. Tomaremos el enfoque simple y ocultaremos sus problemas asegurando que la animación avance en pasos pequeños.

Este método nos indica si un rectángulo (especificado por una posición y un tamaño) toca un elemento de rejilla de un tipo dado.

```
Level.prototype.touches = function(pos, size, type) {  
  let xStart = Math.floor(pos.x);  
  let xEnd = Math.ceil(pos.x + size.x);  
  let yStart = Math.floor(pos.y);  
  let yEnd = Math.ceil(pos.y + size.y);  
  
  for (let y = yStart; y < yEnd; y++) {  
    for (let x = xStart; x < xEnd; x++) {  
      let isOutside = x < 0 || x >= this.width ||  
                     y < 0 || y >= this.height;  
      let here = isOutside ? "wall" : this.rows[y][x];  
      if (here == type) return true;  
    }  
  }  
  return false;  
};
```

El método calcula el conjunto de cuadrados de rejilla con los que el cuerpo se superpone utilizando `Math.floor` y `Math.ceil` en sus coordenadas. Recuerda que los cuadrados de la rejilla son de tamaño 1 por 1 unidad. Al redondear los lados de un cuadro hacia arriba y hacia abajo, obtenemos el rango de cuadrados de fondo que el cuadro toca.



Recorremos el bloque de cuadrados de rejilla encontrado al redondear las coordenadas y devolvemos `true` cuando se encuentra un cuadro coincidente. Los cuadrados fuera del nivel siempre se

tratan como "wall" para asegurar que el jugador no pueda salir del mundo y que no intentemos leer fuera de los límites de nuestra matriz rows.

El método update de estado utiliza touches para determinar si el jugador está tocando lava.

```
State.prototype.update = function(time, keys) {
  let actors = this.actors
    .map(actor => actor.update(time, this, keys));
  let newState = new State(this.level, actors, this.status);

  if (newState.status !== "playing") return newState;

  let player = newState.player;
  if (this.level.touches(player.pos, player.size, "lava")) {
    return new State(this.level, actors, "lost");
  }

  for (let actor of actors) {
    if (actor !== player && overlap(actor, player)) {
      newState = actor.collide(newState);
    }
  }
  return newState;
};
```

El método recibe un paso de tiempo y una estructura de datos que le indica qué teclas se mantienen presionadas. Lo primero que hace es llamar al método update en todos los actores, produciendo un array de actores actualizados. Los actores también reciben el paso de tiempo, las teclas y el estado, para que puedan basar su actualización en esos valores. Solo el jugador realmente lee las teclas, ya que es el único actor controlado por el teclado.

Si el juego ya ha terminado, no es necesario realizar más procesamiento (no se puede ganar el juego después de haber perdido, o viceversa). De lo contrario, el método prueba si el jugador está tocando lava de fondo. Si es así, se pierde el juego y hemos terminado. Finalmente, si el juego sigue en curso, verifica si algún otro actor se superpone al jugador. La superposición entre actores se detecta con la función `overlap`. Toma dos objetos actor y devuelve `true` cuando se tocan, lo cual sucede cuando se superponen tanto a lo largo del eje x como a lo largo del eje y.

```
function overlap(actor1, actor2) {  
  return actor1.pos.x + actor1.size.x > actor2.pos.x &&  
    actor1.pos.x < actor2.pos.x + actor2.size.x &&  
    actor1.pos.y + actor1.size.y > actor2.pos.y &&  
    actor1.pos.y < actor2.pos.y + actor2.size.y;  
}
```

Si algún actor se superpone, su método `collide` tiene la oportunidad de actualizar el estado. Tocar un actor de lava establece el estado del juego en `"lost"`. Las monedas desaparecen cuando las tocas y establecen el estado en `"won"` cuando son la última moneda del nivel.

```
Lava.prototype.collide = function(state) {  
  return new State(state.level, state.actors, "lost");  
};  
  
Coin.prototype.collide = function(state) {  
  let filtered = state.actors.filter(a => a !== this);  
  let status = state.status;  
  if (!filtered.some(a => a.type === "coin")) status = "won";  
  return new State(state.level, filtered, status);  
};
```

ACTUALIZACIONES DE ACTORES

Los métodos `update` de los objetos actor toman como argumentos el paso de tiempo, el objeto de estado y un objeto `keys`. El de tipo actor Lava ignora el objeto `keys`.

```
Lava.prototype.update = function(time, state) {  
  let newPos = this.pos.plus(this.speed.times(time));  
  if (!state.level.touches(newPos, this.size, "wall")) {  
    return new Lava(newPos, this.speed, this.reset);  
  } else if (this.reset) {  
    return new Lava(this.reset, this.speed, this.reset);  
  } else {  
    return new Lava(this.pos, this.speed.times(-1));  
  }  
};
```

Este método `update` calcula una nueva posición agregando el producto del paso de tiempo y la velocidad actual a su posición anterior. Si no hay obstáculos que bloqueen esa nueva posición, se mueve allí. Si hay un obstáculo, el comportamiento depende del tipo de bloque de lava—la lava goteante tiene una posición de `reset` a la que regresa cuando golpea algo. La lava rebotante invierte su velocidad multiplicándola por `-1` para que comience a moverse en la dirección opuesta.

Las monedas utilizan su método `update` para balancearse. Ignoran las colisiones con la cuadrícula ya que simplemente se balancean dentro de su propio cuadrado.

```
const wobbleSpeed = 8, wobbleDist = 0.07;  
  
Coin.prototype.update = function(time) {  
  let wobble = this.wobble + time * wobbleSpeed;  
  let wobblePos = Math.sin(wobble) * wobbleDist;  
  return new Coin(this.basePos.plus(new Vec(0, wobblePos)),
```

```
        this.basePos, wobble);  
    };
```

La propiedad `wobble` se incrementa para hacer un seguimiento del tiempo y luego se utiliza como argumento para `Math.sin` para encontrar la nueva posición en la onda. La posición actual de la moneda se calcula a partir de su posición base y un desplazamiento basado en esta onda.

Eso deja al jugador en sí. El movimiento del jugador se maneja por separado por eje porque golpear el suelo no debería impedir el movimiento horizontal, y golpear una pared no debería detener el movimiento de caída o de salto.

```
const playerXSpeed = 7;  
const gravity = 30;  
const jumpSpeed = 17;  
  
Player.prototype.update = function(time, state, keys) {  
    let xSpeed = 0;  
    if (keys.ArrowLeft) xSpeed -= playerXSpeed;  
    if (keys.ArrowRight) xSpeed += playerXSpeed;  
    let pos = this.pos;  
    let movedX = pos.plus(new Vec(xSpeed * time, 0));  
    if (!state.level.touches(movedX, this.size, "wall")) {  
        pos = movedX;  
    }  
  
    let ySpeed = this.speed.y + time * gravity;  
    let movedY = pos.plus(new Vec(0, ySpeed * time));  
    if (!state.level.touches(movedY, this.size, "wall")) {  
        pos = movedY;  
    } else if (keys.ArrowUp && ySpeed > 0) {  
        ySpeed = -jumpSpeed;  
    } else {  
        ySpeed = 0;  
    }  
}
```

```
    return new Player(pos, new Vec(xSpeed, ySpeed));  
};
```

El movimiento horizontal se calcula en función del estado de las teclas de flecha izquierda y derecha. Cuando no hay una pared bloqueando la nueva posición creada por este movimiento, se utiliza. De lo contrario, se mantiene la posición anterior.

El movimiento vertical funciona de manera similar pero tiene que simular saltos y gravedad. La velocidad vertical del jugador (*ySpeed*) se acelera primero para tener en cuenta la gravedad.

Comprobamos las paredes nuevamente. Si no golpeamos ninguna, se usa la nueva posición. Si *hay* una pared, hay dos posibles resultados. Cuando se presiona la flecha hacia arriba *y* estamos bajando (lo que significa que lo que golpeamos está debajo de nosotros), la velocidad se establece en un valor negativo relativamente grande. Esto hace que el jugador salte. Si ese no es el caso, el jugador simplemente chocó con algo y la velocidad se establece en cero.

La fuerza de la gravedad, la velocidad de salto y otras constantes en el juego se determinaron simplemente probando algunos números y viendo cuáles se sentían correctos. Puedes experimentar con ellos.

SEGUIMIENTO DE TECLAS

Para un juego como este, no queremos que las teclas tengan efecto una vez por pulsación de tecla. Más bien, queremos que su efecto (mover la figura del jugador) se mantenga activo mientras se mantienen presionadas.

Necesitamos configurar un controlador de teclas que almacene el estado actual de las teclas de flecha izquierda, derecha y arriba. También queremos llamar a `preventDefault` para esas teclas para que no terminen desplazando la página.

La siguiente función, al darle un array de nombres de teclas, devolverá un objeto que sigue la posición actual de esas teclas. Registra controladores de eventos para eventos `"keydown"` y `"keyup"` y, cuando el código de tecla en el evento está presente en el conjunto de códigos que está siguiendo, actualiza el objeto.

```
function trackKeys(keys) {
  let down = Object.create(null);
  function track(event) {
    if (keys.includes(event.key)) {
      down[event.key] = event.type == "keydown";
      event.preventDefault();
    }
  }
  window.addEventListener("keydown", track);
  window.addEventListener("keyup", track);
  return down;
}

const arrowKeys = trackKeys(["ArrowLeft", "ArrowRight",
  "ArrowUp"]);
```

La misma función manejadora se utiliza para ambos tipos de eventos. Esta función examina la propiedad `type` del objeto de evento para determinar si el estado de la tecla debe actualizarse a verdadero (`"keydown"`) o falso (`"keyup"`).

EJECUTANDO EL JUEGO

La función `requestAnimationFrame`, que vimos en [Capítulo 14](#), proporciona una buena forma de animar un juego. Pero su interfaz es bastante primitiva, ya que su uso requiere que llevemos un registro del momento en que se llamó a nuestra función la última vez y llamemos a `requestAnimationFrame` nuevamente después de cada fotograma.

Vamos a definir una función auxiliar que envuelva todo eso en una interfaz conveniente y nos permita simplemente llamar a `runAnimation`, dándole una función que espera una diferencia de tiempo como argumento y dibuja un solo fotograma. Cuando la función de fotograma devuelve el valor `false`, la animación se detiene.

```
function runAnimation(frameFunc) {  
  let lastTime = null;  
  function frame(time) {  
    if (lastTime != null) {  
      let timeStep = Math.min(time - lastTime, 100) / 1000;  
      if (frameFunc(timeStep) === false) return;  
    }  
    lastTime = time;  
    requestAnimationFrame(frame);  
  }  
  requestAnimationFrame(frame);  
}
```

He establecido un paso de fotograma máximo de 100 milisegundos (una décima parte de un segundo). Cuando la pestaña del navegador o la ventana con nuestra página está oculta, las llamadas a `requestAnimationFrame` se suspenden hasta que la pestaña o la ventana se vuelva a mostrar. En este caso, la diferencia entre `lastTime` y `time` será todo el tiempo en el que la página estuvo oculta. Avanzar el juego tanto en un solo paso se vería ridículo y

podría causar efectos secundarios extraños, como que el jugador caiga a través del suelo.

La función también convierte los pasos de tiempo a segundos, que son una cantidad más fácil de entender que los milisegundos.

La función `runLevel` toma un objeto `Level` y un constructor de `display` y devuelve una promesa. Muestra el nivel (en `document.body`) y permite al usuario jugar a través de él. Cuando el nivel termina (perdido o ganado), `runLevel` espera un segundo más (para que el usuario vea qué sucede), luego borra la pantalla, detiene la animación y resuelve la promesa con el estado final del juego.

```
function runLevel(level, Display) {
  let display = new Display(document.body, level);
  let state = State.start(level);
  let ending = 1;
  return new Promise(resolve => {
    runAnimation(time => {
      state = state.update(time, arrowKeys);
      display.syncState(state);
      if (state.status == "playing") {
        return true;
      } else if (ending > 0) {
        ending -= time;
        return true;
      } else {
        display.clear();
        resolve(state.status);
        return false;
      }
    });
  });
}
```

Un juego es una secuencia de niveles. Cada vez que el jugador muere, el nivel actual se reinicia. Cuando se completa un nivel, pasamos al siguiente nivel. Esto se puede expresar mediante la siguiente función, que toma un array de planes de nivel (cadenas) y un constructor de display:

```
async function runGame(plans, Display) {
  for (let level = 0; level < plans.length;) {
    let status = await runLevel(new Level(plans[level]),
                                Display);
    if (status == "ganado") level++;
  }
  console.log("¡Has ganado!");
}
```

Debido a que hicimos que `runLevel` devuelva una promesa, `runGame` puede escribirse utilizando una función `async`, como se muestra en [Capítulo 11](#). Devuelve otra promesa, que se resuelve cuando el jugador termina el juego.

Hay un conjunto de planes de niveles disponibles en el enlace `GAME_LEVELS` en el [sandbox de este capítulo](#) (<https://eloquentjavascript.net/code#16>). Esta página los alimenta a `runGame`, comenzando un juego real.

```
<link rel="stylesheet" href="css/game.css">

<body>
  <script>
    runGame(GAME_LEVELS, DOMDisplay);
  </script>
</body>
```

EJERCICIOS

JUEGO TERMINADO

Es tradicional que los juegos de plataformas hagan que el jugador comience con un número limitado de *vidas* y resten una vida cada vez que mueren. Cuando el jugador se queda sin vidas, el juego se reinicia desde el principio.

Ajusta `runGame` para implementar vidas. Haz que el jugador comience con tres vidas. Muestra el número actual de vidas (usando `console.log`) cada vez que comienza un nivel.

PAUSAR EL JUEGO

Haz posible pausar y despausar el juego presionando la tecla Esc.

Esto se puede hacer cambiando la función `runLevel` para configurar un manejador de eventos de teclado que interrumpa o reanude la animación cada vez que se presiona la tecla Esc.

La interfaz de `runAnimation` puede no parecer adecuada para esto a primera vista, pero lo es si reorganizas la forma en que `runLevel` la llama.

Cuando tengas eso funcionando, hay algo más que podrías intentar. La forma en que hemos estado registrando los controladores de eventos de teclado es algo problemática. El objeto `arrowKeys` es actualmente una asignación global, y sus controladores de eventos se mantienen incluso cuando no hay ningún juego en ejecución. Podrías decir que *escapan* de nuestro sistema. Amplía `trackKeys` para proporcionar una forma de anular el registro de sus controladores y

luego cambia `runLevel` para registrar sus controladores cuando comienza y desregistrarlos nuevamente cuando termine.

UN MONSTRUO

Es tradicional que los juegos de plataformas tengan enemigos a los que puedes saltar encima para derrotar. Este ejercicio te pide que agregues un tipo de actor así al juego.

Lo llamaremos monstruo. Los monstruos se mueven solo horizontalmente. Puedes hacer que se muevan en la dirección del jugador, que reboten de un lado a otro como lava horizontal, o tengan cualquier patrón de movimiento que desees. La clase no tiene que manejar caídas, pero debe asegurarse de que el monstruo no atravesase paredes.

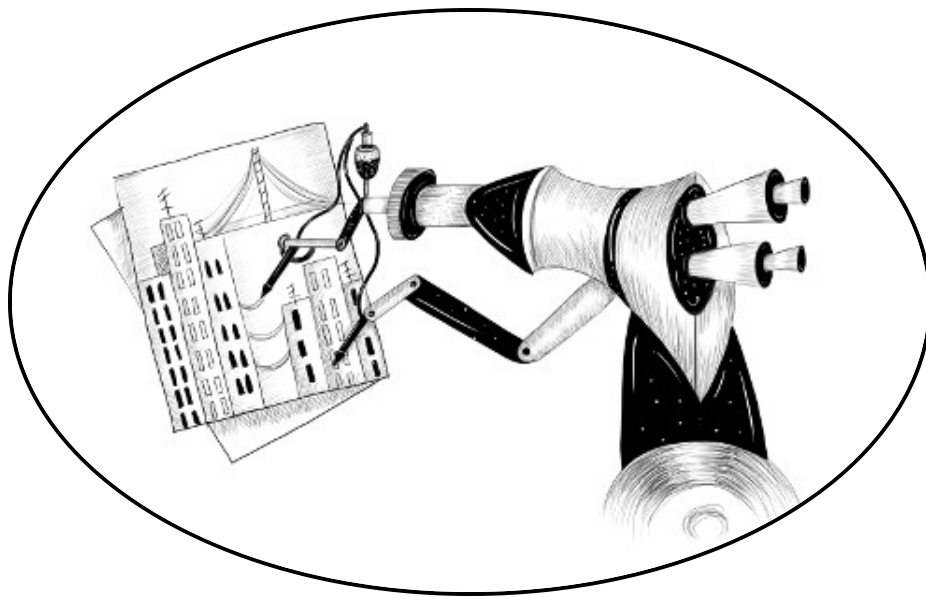
Cuando un monstruo toca al jugador, el efecto depende de si el jugador está saltando encima de ellos o no. Puedes aproximarlo comprobando si el final del jugador está cerca de la parte superior del monstruo. Si este es el caso, el monstruo desaparece. Si no, el juego se pierde.

DIBUJANDO EN CANVAS

```
{{meta {load_files: ["code/chapter/16_game.js", "code/levels.js",  
"code/stop_keys.js", "code/chapter/17_canvas.js"], zip: "html  
include=["img/player.png", "img/sprites.png"]"}}
```

“Dibujar es engañar.”

—M.C. Escher, *citado por Bruno Ernst en El Espejo Mágico de M.C. Escher*



Los navegadores nos ofrecen varias formas de mostrar gráficos. La forma más simple es usar estilos para posicionar y colorear elementos DOM regulares. Esto puede llevarnos bastante lejos, como mostró el juego en el [capítulo anterior](#). Al agregar imágenes de fondo parcialmente transparentes a los nodos, podemos hacer que se vean

exactamente como queremos. Incluso es posible rotar o sesgar nodos con el estilo `transform`.

Pero estaríamos utilizando el DOM para algo para lo que no fue diseñado originalmente. Algunas tareas, como dibujar una línea entre puntos arbitrarios, son extremadamente incómodas de hacer con elementos HTML regulares.

Hay dos alternativas. La primera es basada en el DOM pero utiliza *Gráficos Vectoriales Escalables* (SVG), en lugar de HTML. Piensa en SVG como un dialecto de marcado de documento que se centra en las formas en lugar de en el texto. Puedes incrustar un documento SVG directamente en un documento HTML o incluirlo con una etiqueta ``.

La segunda alternativa se llama *lienzo*. Un lienzo es un solo elemento DOM que encapsula una imagen. Proporciona una interfaz de programación para dibujar formas en el espacio ocupado por el nodo. La principal diferencia entre un lienzo y una imagen SVG es que en SVG se conserva la descripción original de las formas para que puedan moverse o redimensionarse en cualquier momento. Un lienzo, por otro lado, convierte las formas en píxeles (puntos de color en una cuadrícula) en cuanto se dibujan y no recuerda qué representan estos píxeles. La única forma de mover una forma en un lienzo es borrar el lienzo (o la parte del lienzo alrededor de la forma) y volver a dibujarlo con la forma en una nueva posición.

SVG

Este libro no se adentrará en detalles sobre SVG, pero explicaré brevemente cómo funciona. Al [final del capítulo](#), volveré a los

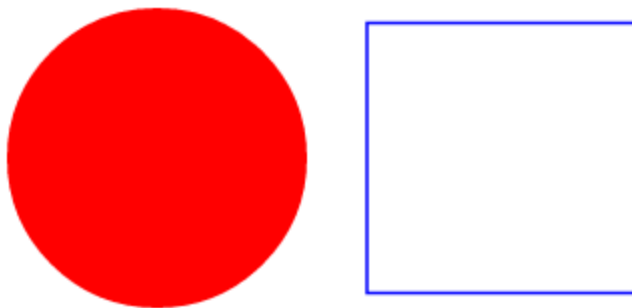
compromisos que debes considerar al decidir qué mecanismo de dibujo es adecuado para una aplicación determinada.

Este es un documento HTML con una sencilla imagen SVG en él:

```
<p>Aquí va HTML normal.</p>
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="50" cx="50" cy="50" fill="red"/>
  <rect x="120" y="5" width="90" height="90"
    stroke="blue" fill="none"/>
</svg>
```

El documento se muestra de la siguiente manera:

Normal HTML here.



Estas etiquetas crean elementos del DOM, al igual que las etiquetas HTML, con las que los scripts pueden interactuar. Por ejemplo, esto cambia el elemento `<circle>` para que se coloree de cian:

```
let circle = document.querySelector("circle");
circle.setAttribute("fill", "cyan");
```

EL ELEMENTO CANVAS

Los gráficos en lienzo pueden ser dibujados en un elemento `<canvas>`. Puedes darle a dicho elemento atributos `width` y

`height` para determinar su tamaño en píxels.

Un lienzo nuevo está vacío, lo que significa que es completamente transparente y por lo tanto se muestra como espacio vacío en el documento.

La etiqueta `<canvas>` está destinada a permitir diferentes estilos de dibujo. Para acceder a una interfaz de dibujo real, primero necesitamos crear un *contexto*, un objeto cuyos métodos proporcionan la interfaz de dibujo. Actualmente existen tres estilos de dibujo ampliamente compatibles: "2d" para gráficos bidimensionales, "webgl" para gráficos tridimensionales a través de la interfaz OpenGL, y "webgpu", una alternativa más moderna y flexible a WebGL.

Este libro no discutirá WebGL ni WebGPU—nos mantendremos en dos dimensiones. Pero si estás interesado en gráficos tridimensionales, te animo a investigar sobre WebGPU. Proporciona una interfaz directa al hardware gráfico y te permite renderizar escenas incluso complicadas de manera eficiente, utilizando JavaScript.

Creas un contexto con el método `getContext` en el elemento DOM `<canvas>`.

```
<p>Antes del lienzo.</p>
<canvas width="120" height="60"></canvas>
<p>Después del lienzo.</p>
<script>
  let canvas = document.querySelector("canvas");
  let context = canvas.getContext("2d");
  context.fillStyle = "red";
```

```
context.fillRect(10, 10, 100, 50);  
</script>
```

Después de crear el objeto de contexto, el ejemplo dibuja un rectángulo rojo de 100 píxeles de ancho y 50 píxeles de alto, con su esquina superior izquierda en las coordenadas (10,10).

Before canvas.



After canvas.

Al igual que en HTML (y SVG), el sistema de coordenadas que utiliza el lienzo sitúa el (0,0) en la esquina superior izquierda, y el eje y-positivo va hacia abajo desde allí. Por lo tanto, (10,10) está 10 píxeles abajo y a la derecha de la esquina superior izquierda.

LÍNEAS Y SUPERFICIES

En la interfaz de lienzo, una forma puede ser *rellenada*, lo que significa que su área recibe un color o patrón determinado, o puede ser *trazada*, lo que significa que se dibuja una línea a lo largo de su borde. La misma terminología se utiliza en SVG.

El método `fillRect` rellena un rectángulo. Primero toma las coordenadas x e y de la esquina superior izquierda del rectángulo,

luego su ancho y finalmente su altura. Un método similar llamado `strokeRect` dibuja el contorno de un rectángulo.

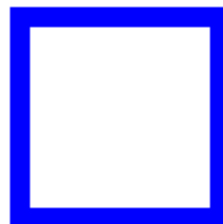
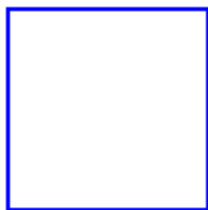
Ninguno de los métodos toma más parámetros. El color del relleno, el grosor del trazo, y demás, no son determinados por un argumento del método, como podrías esperar razonablemente, sino por propiedades del objeto contexto.

La propiedad `fillStyle` controla la forma en que se rellenan las formas. Puede establecerse como una cadena que especifica un color, utilizando la notación de color utilizada por CSS.

La propiedad `strokeStyle` funciona de manera similar, pero determina el color utilizado para una línea contorneada. El ancho de esa línea se determina mediante la propiedad `lineWidth`, que puede contener cualquier número positivo.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.strokeStyle = "blue";
  cx.strokeRect(5, 5, 50, 50);
  cx.lineWidth = 5;
  cx.strokeRect(135, 5, 50, 50);
</script>
```

Este código dibuja dos cuadrados azules, usando una línea más gruesa para el segundo.



Cuando no se especifica ningún atributo `width` o `height`, como en el ejemplo, un elemento `canvas` obtiene un ancho predeterminado de 300 píxeles y una altura de 150 píxeles.

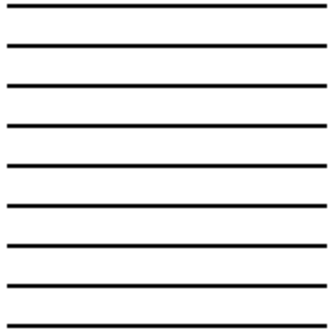
CAMINOS

Un camino es una secuencia de líneas. La interfaz del `canvas` 2D toma un enfoque peculiar para describir un camino. Se realiza completamente a través de efectos secundarios. Los caminos no son valores que se puedan almacenar y pasar. En su lugar, si deseas hacer algo con un camino, haces una secuencia de llamadas a métodos para describir su forma.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  for (let y = 10; y < 100; y += 10) {
    cx.moveTo(10, y);
    cx.lineTo(90, y);
  }
  cx.stroke();
</script>
```

Este ejemplo crea un camino con varios segmentos horizontales de línea y luego lo traza usando el método `stroke`. Cada segmento creado con `lineTo` comienza en la posición *actual* del camino. Esa posición suele ser el final del último segmento, a menos que se haya llamado a `moveTo`. En ese caso, el siguiente segmento comenzaría en la posición pasada a `moveTo`.

El camino descrito por el programa anterior se ve así:



Cuando se rellena un camino (usando el método `fill`), cada forma se llena por separado. Un camino puede contener múltiples formas—cada movimiento de `moveTo` inicia una nueva forma. Pero el camino necesita estar *cerrado* (significando que su inicio y final están en la misma posición) antes de poder ser relleno. Si el camino aún no está cerrado, se agrega una línea desde su final hasta su inicio, y se rellena la forma encerrada por el camino completado.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(50, 10);
  cx.lineTo(10, 70);
  cx.lineTo(90, 70);
  cx.fill();
</script>
```

Este ejemplo dibuja un triángulo relleno. Ten en cuenta que solo se dibujan explícitamente dos de los lados del triángulo. El tercero, desde la esquina inferior derecha de regreso a la parte superior, se da por implícito y no estaría allí cuando se traze el recorrido.



También puedes usar el método `closePath` para cerrar explícitamente un recorrido agregando un segmento real line de vuelta al inicio del recorrido. Este segmento se dibuja cuando se traza el recorrido.

CURVAS

Un recorrido también puede contener líneas curvadas. Lamentablemente, estas son un poco más complicadas de dibujar.

El método `quadraticCurveTo` dibuja una curva hacia un punto dado. Para determinar la curvatura de la línea, el método recibe un punto de control así como un punto de destino. Imagina este punto de control como *atrayendo* la línea, dándole su curva. La línea no pasará por el punto de control, pero su dirección en los puntos de inicio y fin será tal que una línea recta en esa dirección apuntaría hacia el punto de control. El siguiente ejemplo ilustra esto:

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(10, 90);
  // control=(60,10) meta=(90,90)
  cx.quadraticCurveTo(60, 10, 90, 90);
  cx.lineTo(60, 10);
  cx.closePath();
```

```
cx.stroke();  
</script>
```

Produce un recorrido que se ve así:



Dibujamos una curva cuadrática de izquierda a derecha, con (60,10) como punto de control, y luego dibujamos dos segmentos line que pasan por ese punto de control y vuelven al inicio de la línea. El resultado se asemeja a un emblema de *Star Trek*. Puedes ver el efecto del punto de control: las líneas que salen de las esquinas inferiores comienzan en la dirección del punto de control y luego se curvan hacia su objetivo.

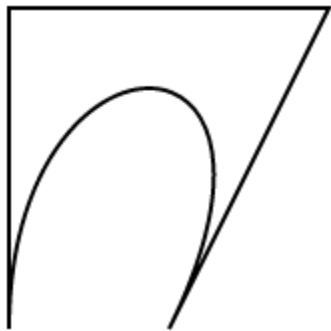
El método `bezierCurveTo` dibuja un tipo de curva similar. En lugar de un único punto de control, este tiene dos—uno para cada uno de los extremos de la línea. Aquí hay un boceto similar para ilustrar el comportamiento de dicha curva:

```
<canvas></canvas>  
<script>  
  let cx = document.querySelector("canvas").getContext("2d");  
  cx.beginPath();  
  cx.moveTo(10, 90);  
  // control1=(10,10) control2=(90,10) meta=(50,90)  
  cx.bezierCurveTo(10, 10, 90, 10, 50, 90);  
  cx.lineTo(90, 10);  
</script>
```



```
cx.lineTo(10, 10);  
cx.closePath();  
cx.stroke();  
</script>
```

Los dos puntos de control especifican la dirección en ambos extremos de la curva. Cuanto más separados estén de su punto correspondiente, más la curva “abultará” en esa dirección.



curves como estas pueden ser difíciles de trabajar, no siempre es claro cómo encontrar los control points que proporcionan la forma que estás buscando. A veces puedes calcularlos y a veces simplemente tendrás que encontrar un valor adecuado mediante prueba y error.

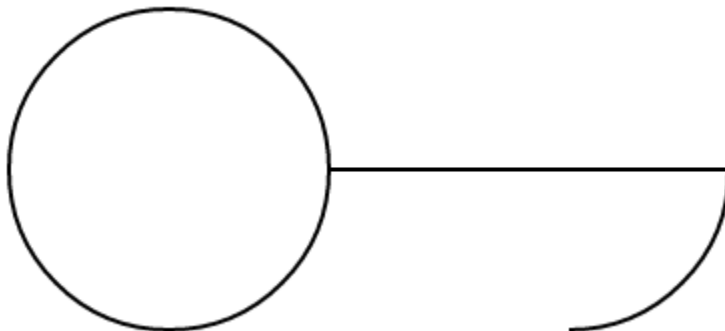
El método `arc` es una forma de dibujar una línea que se curva a lo largo del borde de un círculo. Toma un par de coordenadas para el centro del arco, un radio, y luego un ángulo de inicio y un ángulo final.

Estos últimos dos parámetros permiten dibujar solo parte del círculo. Los ángulos se miden en radianes, no en grados. Esto significa que un círculo completo tiene un ángulo de 2π , o $2 * \text{Math.PI}$, que es aproximadamente 6.28. El ángulo comienza a contar en el punto a la derecha del centro del círculo y va en sentido

horario desde allí. Puedes usar un inicio de 0 y un final mayor que 2π (por ejemplo, 7) para dibujar un círculo completo.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  // centro=(50,50) radio=40 ángulo=0 a 7
  cx.arc(50, 50, 40, 0, 7);
  // centro=(150,50) radio=40 ángulo=0 a  $\frac{1}{2}\pi$ 
  cx.arc(150, 50, 40, 0, 0.5 * Math.PI);
  cx.stroke();
</script>
```

La imagen resultante contiene una línea desde la derecha del círculo completo (primer llamado a `arc`) hasta la derecha del cuarto del círculo (segundo llamado). Al igual que otros métodos de dibujo de trayectos, una línea dibujada con `arc` está conectada al segmento de trayecto anterior. Puedes llamar a `moveTo` o comenzar un nuevo trayecto para evitar esto.



DIBUJO DE UN DIAGRAMA DE SECTORES

Imagina que acabas de aceptar un trabajo en EconomiCorp, Inc., y tu primera tarea es dibujar un diagrama de sectores de los resultados de la encuesta de satisfacción de los clientes.

El enlace `results` contiene una matriz de objetos que representan las respuestas de la encuesta.

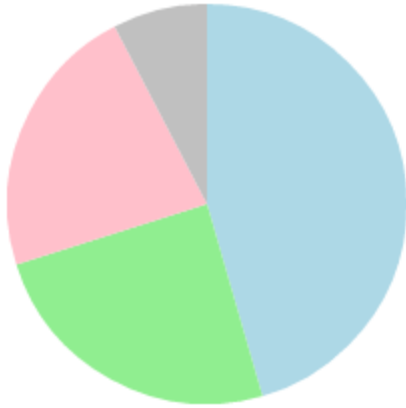
```
const results = [  
  {name: "Satisfecho", count: 1043, color: "lightblue"},  
  {name: "Neutral", count: 563, color: "lightgreen"},  
  {name: "Insatisfecho", count: 510, color: "pink"},  
  {name: "Sin comentario", count: 175, color: "silver"}  
];
```

Para dibujar un diagrama de sectores, dibujamos una serie de sectores circulares, cada uno compuesto por un arco y un par de líneas hacia el centro de ese arco. Podemos calcular el ángulo ocupado por cada arco dividiendo un círculo completo (2π) por el número total de respuestas y luego multiplicando ese número (el ángulo por respuesta) por el número de personas que eligieron una opción determinada.

```
<canvas width="200" height="200"></canvas>  
<script>  
  let cx = document.querySelector("canvas").getContext("2d");  
  let total = results  
    .reduce((sum, {count}) => sum + count, 0);  
  // Comenzar en la parte superior  
  let currentAngle = -0.5 * Math.PI;  
  for (let result of results) {  
    let sliceAngle = (result.count / total) * 2 * Math.PI;  
    cx.beginPath();  
    // centro=100,100, radio=100  
    // desde el ángulo actual, en sentido horario por el ángulo  
del sector  
    cx.arc(100, 100, 100,  
           currentAngle, currentAngle + sliceAngle);  
    currentAngle += sliceAngle;  
    cx.lineTo(100, 100);  
    cx.fillStyle = result.color;  
    cx.fill();  
  }  
</script>
```

```
}  
</script>
```

Esto dibuja el siguiente gráfico:



Pero un gráfico que no nos dice qué significan las porciones no es muy útil. Necesitamos una forma de dibujar texto en el canvas.

TEXTO

Un contexto de dibujo en lienzo 2D proporciona los métodos `fillText` y `strokeText`. Este último puede ser útil para contornear letras, pero generalmente `fillText` es lo que necesitas. Este llenará el contorno del texto dado con el `fillStyle` actual.

```
<canvas></canvas>  
<script>  
  let cx = document.querySelector("canvas").getContext("2d");  
  cx.font = "28px Georgia";  
  cx.fillStyle = "fuchsia";  
  cx.fillText("¡También puedo dibujar texto!", 10, 50);  
</script>
```

Puedes especificar el tamaño, estilo y fuente del texto con la propiedad `font`. Este ejemplo solo da un tamaño de fuente y un

nombre de familia. También es posible agregar `italic` o `bold` al comienzo de la cadena para seleccionar un estilo.

Los dos últimos argumentos de `fillText` y `strokeText` proporcionan la posición en la que se dibuja la fuente. Por defecto, indican la posición del inicio de la línea alfabética del texto, que es la línea en la que las letras “se paran”, sin contar las partes colgantes en letras como la *j* o la *p*. Puedes cambiar la posición horizontal configurando la propiedad `textAlign` en `"end"` o `"center"` y la posición vertical configurando `textBaseline` en `"top"`, `"middle"` o `"bottom"`.

Volveremos a nuestro gráfico circular y al problema de etiquetar las porciones, en los [ejercicios](#) al final del capítulo.

IMÁGENES

En gráficos por computadora, a menudo se hace una distinción entre gráficos *vectoriales* y gráficos *de mapa de bits*. El primero es lo que hemos estado haciendo hasta ahora en este capítulo: especificar una imagen dando una descripción lógica de las formas. Los gráficos de mapa de bits, por otro lado, no especifican formas reales, sino que trabajan con datos de píxel (rasteros de puntos de colores).

El método `drawImage` nos permite dibujar datos de píxel en un canvas. Estos datos de píxel pueden originarse desde un elemento `` o desde otro lienzo. El siguiente ejemplo crea un elemento `` independiente y carga un archivo de imagen en él. Pero no podemos comenzar a dibujar inmediatamente desde esta imagen porque es posible que el navegador aún no la haya cargado. Para

manejar esto, registramos un controlador de eventos "load" y hacemos el dibujo después de que la imagen se haya cargado.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/hat.png";
  img.addEventListener("load", () => {
    for (let x = 10; x < 200; x += 30) {
      cx.drawImage(img, x, 10);
    }
  });
</script>
```

Por defecto, `drawImage` dibujará la imagen a su tamaño original. También se le pueden proporcionar dos argumentos adicionales para establecer un ancho y alto diferente.

Cuando se utilizan *nueve* argumentos en `drawImage`, se puede usar para dibujar solo un fragmento de una imagen. Los argumentos segundo a quinto indican el rectángulo (x, y, ancho y alto) en la imagen de origen que se debería copiar, y los argumentos sexto a noveno indican el rectángulo (en el lienzo) en el cual se debería copiar.

Esto se puede utilizar para empaquetar varios *sprites* (elementos de imagen) en un único archivo de imagen y luego dibujar solo la parte que se necesita. Por ejemplo, tenemos esta imagen que contiene un personaje de juego en múltiples poses:



Alternando qué pose dibujamos, podemos mostrar una animación que parece un personaje caminando.

Para animar una imagen en un lienzo, el método `clearRect` es útil. Se asemeja a `fillRect`, pero en lugar de colorear el rectángulo, lo vuelve transparente, eliminando los píxeles dibujados anteriormente.

Sabemos que cada *sprite*, cada subimagen, tiene un ancho de 24 píxeles y una altura de 30 píxeles. El siguiente código carga la imagen y luego establece un intervalo (temporizador repetido) para dibujar el siguiente frame:

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/player.png";
  let spriteW = 24, spriteH = 30;
  img.addEventListener("load", () => {
    let ciclo = 0;
    setInterval(() => {
      cx.clearRect(0, 0, spriteW, spriteH);
      cx.drawImage(img,
                    // rectángulo de origen
                    ciclo * spriteW, 0, spriteW, spriteH,
                    // rectángulo de destino
                    0, 0, spriteW, spriteH);
      ciclo = (ciclo + 1) % 8;
    }, 120);
  });
</script>
```

El enlace `ciclo` sigue nuestra posición en la animación. En cada frame, se incrementa y luego se recorta de nuevo al rango de 0 a 7 usando el operador de resto. Este enlace se utiliza luego para calcular la coordenada x que tiene el sprite para la pose actual en la imagen.

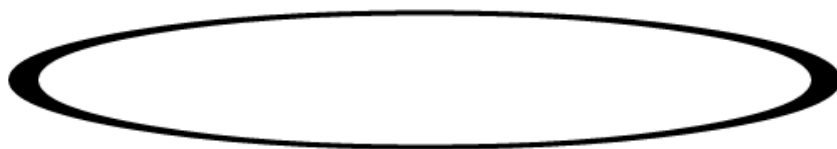
TRANSFORMACIÓN

Pero, ¿qué pasa si queremos que nuestro personaje camine hacia la izquierda en lugar de hacia la derecha? Podríamos dibujar otro conjunto de sprites, por supuesto. Pero también podemos instruir al lienzo para que dibuje la imagen en sentido contrario.

Llamar al método `scale` hará que todo lo que se dibuje después de él se escale. Este método toma dos parámetros, uno para establecer una escala horizontal y otro para establecer una escala vertical.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.scale(3, .5);
  cx.beginPath();
  cx.arc(50, 50, 40, 0, 7);
  cx.lineWidth = 3;
  cx.stroke();
</script>
```

Debido a la llamada a `scale`, el círculo se dibuja tres veces más ancho y la mitad de alto.



Escalar hará que todo en la imagen dibujada, incluyendo el grosor de línea, se estire o se comprima como se especifique. Escalar por una cantidad negativa volteará la imagen. La volteadura ocurre alrededor del punto (0,0), lo que significa que también volteará la dirección del sistema de coordenadas. Cuando se aplica una escala horizontal de

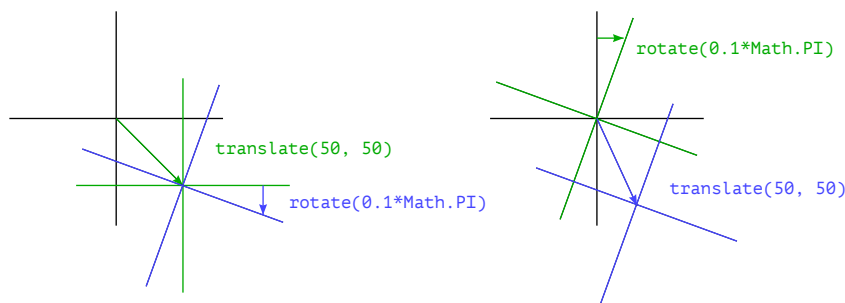
-1, una forma dibujada en la posición x 100 terminará en lo que solía ser la posición -100.

Así que para voltear una imagen, no podemos simplemente agregar `cx.scale(-1, 1)` antes de la llamada a `drawImage` porque eso movería nuestra imagen fuera del lienzo, donde no sería visible.

Podrías ajustar las coordenadas dadas a `drawImage` para compensar esto dibujando la imagen en la posición x -50 en lugar de 0. Otra solución, que no requiere que el código que hace el dibujo sepa sobre el cambio de escala, es ajustar el eje alrededor del cual ocurre el escalado.

Hay varios otros métodos además de `scale` que influyen en el sistema de coordenadas de un lienzo. Puedes rotar formas dibujadas posteriormente con el método `rotate` y moverlas con el método `translate`. Lo interesante—y confuso—es que estas transformaciones *se apilan*, lo que significa que cada una ocurre relativa a las transformaciones anteriores.

Entonces, si traducimos por 10 píxeles horizontales dos veces, todo se dibujará 20 píxeles a la derecha. Si primero movemos el centro del sistema de coordenadas a (50,50) y luego rotamos por 20 grados (aproximadamente 0.1π radianes), esa rotación ocurrirá *alrededor* del punto (50,50).

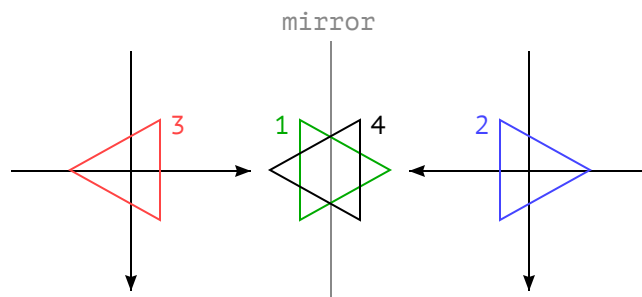


Pero si *primero* rotamos 20 grados y *luego* traducimos por (50,50), la traducción ocurrirá en el sistema de coordenadas rotado y producirá una orientación diferente. El orden en el que se aplican las transformaciones es importante.

Para voltear una imagen alrededor de la línea vertical en una posición x dada, podemos hacer lo siguiente:

```
function flipHorizontally(context, around) {  
    context.translate(around, 0);  
    context.scale(-1, 1);  
    context.translate(-around, 0);  
}
```

Movemos el eje y a donde queremos que esté nuestro espejo, aplicamos el efecto de espejo y finalmente devolvemos el eje y a su lugar adecuado en el universo espejado. La siguiente imagen explica por qué esto funciona:



Esto muestra los sistemas de coordenadas antes y después del espejo a través de la línea central. Los triángulos están numerados para ilustrar cada paso. Si dibujamos un triángulo en una posición x positiva, por defecto estaría en el lugar donde se encuentra el triángulo 1. Una llamada a `flipHorizontally` primero realiza una traslación a la derecha, lo que nos lleva al triángulo 2. Luego

escala, volteando el triángulo a la posición 3. Esto no es donde debería estar, si estuviera reflejado en la línea dada. La segunda llamada a `translate` corrige esto, “cancela” la traslación inicial y hace que el triángulo 4 aparezca exactamente donde debería.

Ahora podemos dibujar un personaje espejado en la posición (100,0) volteando el mundo alrededor del centro vertical del personaje.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/jugador.png";
  let spriteW = 24, spriteH = 30;
  img.addEventListener("load", () => {
    flipHorizontally(cx, 100 + spriteW / 2);
    cx.drawImage(img, 0, 0, spriteW, spriteH,
                  100, 0, spriteW, spriteH);
  });
</script>
```

ALMACENANDO Y ELIMINANDO TRANSFORMACIONES

Las transformaciones permanecen. Todo lo que dibujemos después de ese personaje espejado también estará reflejado. Eso podría ser inconveniente.

Es posible guardar la transformación actual, hacer algunos dibujos y transformaciones, y luego restaurar la antigua transformación. Esto suele ser lo apropiado para una función que necesita transformar temporalmente el sistema de coordenadas. Primero, guardamos cualquier transformación que estuviera utilizando el código que llamó a la función. Luego, la función realiza su tarea, agregando más

transformaciones sobre la transformación actual. Finalmente, volvemos a la transformación con la que comenzamos.

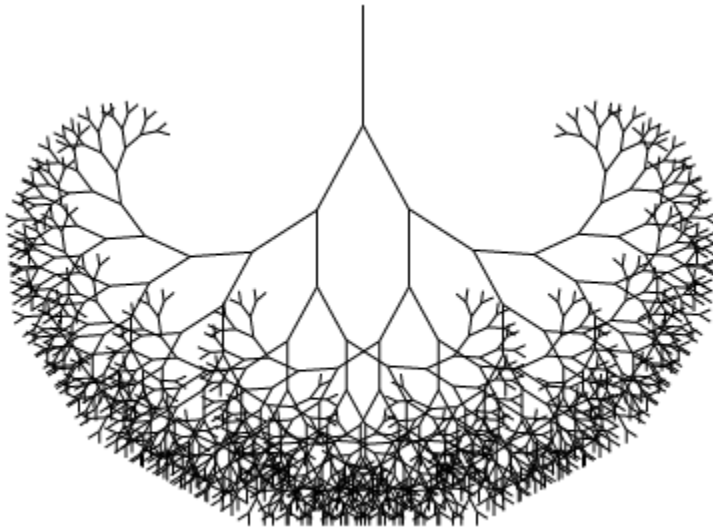
Los métodos `save` y `restore` en el contexto 2D del lienzo hacen este manejo de transformaciones. Conceptualmente mantienen una pila de estados de transformación. Cuando llamas a `save`, el estado actual se apila, y cuando llamas a `restore`, se elimina el estado de la cima de la pila y se usa como la transformación actual del contexto. También puedes llamar a `resetTransform` para restablecer completamente la transformación.

La función `branch` en el siguiente ejemplo ilustra lo que puedes hacer con una función que cambia la transformación y luego llama a una función (en este caso a sí misma), que continúa dibujando con la transformación dada. Esta función dibuja una forma parecida a un árbol dibujando una línea, moviendo el centro del sistema de coordenadas al final de la línea, y llamándose a sí misma dos veces, primero rotada a la izquierda y luego rotada a la derecha. Cada llamada reduce la longitud de la rama dibujada, y la recursividad se detiene cuando la longitud desciende por debajo de 8.

```
<canvas width="600" height="300"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  function branch(length, angle, scale) {
    cx.fillRect(0, 0, 1, length);
    if (length < 8) return;
    cx.save();
    cx.translate(0, length);
    cx.rotate(-angle);
    branch(length * scale, angle, scale);
    cx.rotate(2 * angle);
    branch(length * scale, angle, scale);
    cx.restore();
  }
</script>
```

```
}  
cx.translate(300, 0);  
branch(60, 0.5, 0.8);  
</script>
```

El resultado es un fractal simple.



Si las llamadas a `save` y `restore` no estuvieran allí, la segunda llamada recursiva a `branch` terminaría con la posición y rotación creadas por la primera llamada. No estaría conectada a la rama actual sino más bien a la rama más interna y a la derecha dibujada por la primera llamada. La forma resultante podría ser interesante, pero definitivamente no sería un árbol.

DE VUELTA AL JUEGO

Ahora sabemos lo suficiente sobre el dibujo en canvas para empezar a trabajar en un sistema de display basado en canvas para el juego del [capítulo anterior](#). El nuevo display ya no mostrará solo cajas de colores. En su lugar, usaremos `drawImage` para dibujar imágenes que representen los elementos del juego.

Definimos otro tipo de objeto de display llamado `CanvasDisplay`, que soporta la misma interfaz que `DOMDisplay` del [Capítulo 16](#), es decir, los métodos `syncState` y `clear`.

Este objeto mantiene un poco más de información que `DOMDisplay`. En lugar de utilizar la posición de desplazamiento de su elemento DOM, realiza un seguimiento de su propio viewport, que nos indica qué parte del nivel estamos viendo actualmente. Por último, mantiene una propiedad `flipPlayer` para que incluso cuando el jugador esté quieto, siga mirando en la dirección en la que se movió por última vez.

```
class CanvasDisplay {
  constructor(parent, level) {
    this.canvas = document.createElement("canvas");
    this.canvas.width = Math.min(600, level.width * scale);
    this.canvas.height = Math.min(450, level.height * scale);
    parent.appendChild(this.canvas);
    this.cx = this.canvas.getContext("2d");

    this.flipPlayer = false;

    this.viewport = {
      left: 0,
      top: 0,
      width: this.canvas.width / scale,
      height: this.canvas.height / scale
    };
  }

  clear() {
    this.canvas.remove();
  }
}
```

El método `syncState` primero calcula un nuevo viewport y luego dibuja la escena del juego en la posición adecuada.

```
CanvasDisplay.prototype.syncState = function(state) {
  this.updateViewport(state);
  this.clearDisplay(state.status);
  this.drawBackground(state.level);
  this.drawActors(state.actors);
};
```

A diferencia de `DOMDisplay`, este estilo de visualización **sí** tiene que redibujar el fondo en cada actualización. Debido a que las formas en un lienzo son solo píxeles, una vez que las dibujamos no hay una buena manera de moverlas (o eliminarlas). La única forma de actualizar la visualización en lienzo es borrarla y volver a dibujar la escena. También puede ser que hayamos hecho scroll, lo que requeriría que el fondo esté en una posición diferente.

El método `updateViewport` es similar al método `scrollPlayerIntoView` de `DOMDisplay`. Verifica si el jugador está demasiado cerca del borde de la pantalla y mueve el **viewport** en ese caso.

```
CanvasDisplay.prototype.updateViewport = function(state) {
  let view = this.viewport, margin = view.width / 3;
  let player = state.player;
  let center = player.pos.plus(player.size.times(0.5));

  if (center.x < view.left + margin) {
    view.left = Math.max(center.x - margin, 0);
  } else if (center.x > view.left + view.width - margin) {
    view.left = Math.min(center.x + margin - view.width,
                        state.level.width - view.width);
  }

  if (center.y < view.top + margin) {
    view.top = Math.max(center.y - margin, 0);
  } else if (center.y > view.top + view.height - margin) {
    view.top = Math.min(center.y + margin - view.height,
                        state.level.height - view.height);
  }
}
```

```
}  
};
```

Las llamadas a `Math.max` y `Math.min` aseguran que el **viewport** no termine mostrando espacio fuera del nivel. `Math.max(x, 0)` se asegura de que el número resultante no sea menor que cero.

`Math.min` garantiza de manera similar que un valor se mantenga por debajo de un límite dado.

Al **limpiar** la visualización, usaremos un color ligeramente diferente según si el juego se ha ganado (más brillante) o perdido (más oscuro).

```
CanvasDisplay.prototype.clearDisplay = function(status) {  
  if (status == "won") {  
    this.cx.fillStyle = "rgb(68, 191, 255)";  
  } else if (status == "lost") {  
    this.cx.fillStyle = "rgb(44, 136, 214)";  
  } else {  
    this.cx.fillStyle = "rgb(52, 166, 251)";  
  }  
  this.cx.fillRect(0, 0,  
                    this.canvas.width, this.canvas.height);  
};
```

Para dibujar el fondo, recorreremos los mosaicos que son visibles en el **viewport** actual, utilizando el mismo truco usado en el método `touches` del [capítulo anterior](#).

```
let otherSprites = document.createElement("img");  
otherSprites.src = "img/sprites.png";
```

```
CanvasDisplay.prototype.drawBackground = function(level) {  
  let {left, top, width, height} = this.viewport;  
  let xStart = Math.floor(left);  
  let xEnd = Math.ceil(left + width);
```



```

let yStart = Math.floor(top);
let yEnd = Math.ceil(top + height);

for (let y = yStart; y < yEnd; y++) {
  for (let x = xStart; x < xEnd; x++) {
    let tile = level.rows[y][x];
    if (tile == "empty") continue;
    let screenX = (x - left) * scale;
    let screenY = (y - top) * scale;
    let tileX = tile == "lava" ? scale : 0;
    this.cx.drawImage(otherSprites,
                      tileX, 0, scale, scale,
                      screenX, screenY, scale, scale);
  }
}
};

```

Las casillas que no están vacías se dibujan con `drawImage`. La imagen `otherSprites` contiene las imágenes utilizadas para elementos que no son el jugador. Contiene, de izquierda a derecha, la casilla de pared, la casilla de lava y el sprite de una moneda.



Las casillas de fondo son de 20 por 20 píxeles ya que usaremos la misma escala que en `DOMDisplay`. Por lo tanto, el desplazamiento para las casillas de lava es 20 (el valor del enlace `scale`), y el desplazamiento para las paredes es 0.

No nos molesta esperar a que se cargue la imagen del sprite. Llamar a `drawImage` con una imagen que aún no se ha cargado simplemente no hará nada. Por lo tanto, podríamos no dibujar correctamente el juego durante los primeros cuadros, mientras la imagen aún se está cargando, pero eso no es un problema grave.

Dado que seguimos actualizando la pantalla, la escena correcta aparecerá tan pronto como termine la carga.

El personaje de movimiento que se mostró anteriormente se utilizará para representar al jugador. El código que lo dibuja necesita seleccionar el sprite adecuado y la dirección basándose en el movimiento actual del jugador. Los primeros ocho sprites contienen una animación de caminar. Cuando el jugador se está moviendo a lo largo de una superficie, los recorreremos según el tiempo actual. Queremos cambiar de fotogramas cada 60 milisegundos, por lo que primero dividimos el tiempo por 60. Cuando el jugador está quieto, dibujamos el noveno sprite. Durante los saltos, que se reconocen por el hecho de que la velocidad vertical no es cero, usamos el décimo sprite de la derecha.

Dado que los sprites son ligeramente más anchos que el objeto del jugador—24 en lugar de 16 píxeles para permitir algo de espacio para los pies y los brazos—el método debe ajustar la coordenada x y el ancho por una cantidad dada (`playerXOverlap`).

```
let playerSprites = document.createElement("img");
playerSprites.src = "img/player.png";
const playerXOverlap = 4;

CanvasDisplay.prototype.drawPlayer = function(player, x, y,
                                              width, height){
    width += playerXOverlap * 2;
    x -= playerXOverlap;
    if (player.speed.x != 0) {
        this.flipPlayer = player.speed.x < 0;
    }

    let tile = 8;
    if (player.speed.y != 0) {
        tile = 9;
```

```

    } else if (player.speed.x != 0) {
        tile = Math.floor(Date.now() / 60) % 8;
    }

    this.cx.save();
    if (this.flipPlayer) {
        flipHorizontally(this.cx, x + width / 2);
    }
    let tileX = tile * width;
    this.cx.drawImage(playerSprites, tileX, 0, width, height,
                      x, y, width, height);

    this.cx.restore();
};

```

El método `drawPlayer` es llamado por `drawActors`, el cual es responsable de dibujar todos los actores en el juego.

```

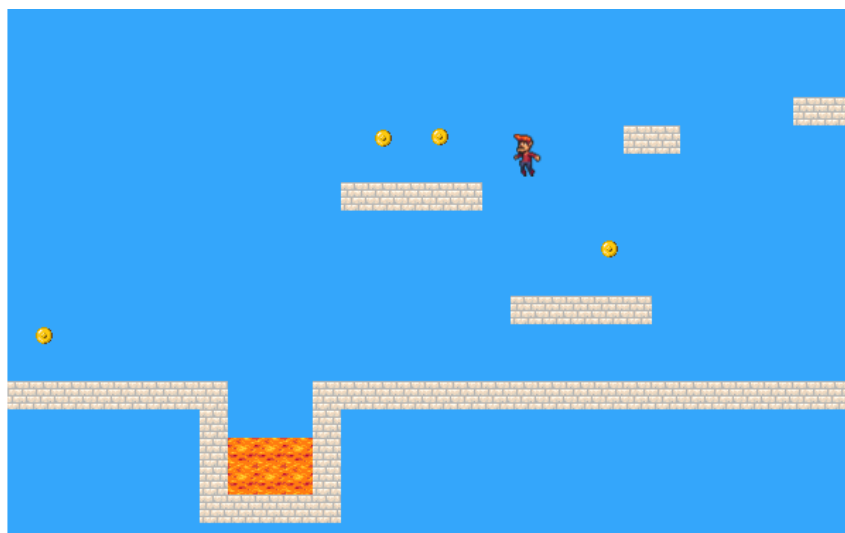
CanvasDisplay.prototype.drawActors = function(actors) {
    for (let actor of actors) {
        let width = actor.size.x * scale;
        let height = actor.size.y * scale;
        let x = (actor.pos.x - this.viewport.left) * scale;
        let y = (actor.pos.y - this.viewport.top) * scale;
        if (actor.type == "player") {
            this.drawPlayer(actor, x, y, width, height);
        } else {
            let tileX = (actor.type == "coin" ? 2 : 1) * scale;
            this.cx.drawImage(otherSprites,
                             tileX, 0, width, height,
                             x, y, width, height);
        }
    }
};

```

Cuando se está dibujando algo que no es el jugador, miramos su tipo para encontrar el desplazamiento del sprite correcto. El tile de lava se encuentra en el desplazamiento 20, y el sprite de la moneda se encuentra en 40 (dos veces `scale`).

Tenemos que restar la posición del viewport al calcular la posición del actor, ya que (0,0) en nuestro canvas corresponde a la esquina superior izquierda del viewport, no a la esquina superior izquierda del nivel. También podríamos haber usado `translate` para esto. De ambas maneras funciona.

Eso concluye el nuevo sistema de display. El juego resultante se ve algo así:



ELECCIÓN DE UNA INTERFAZ GRÁFICA

Por lo tanto, cuando necesitas generar gráficos en el navegador, puedes elegir entre HTML simple, SVG y canvas. No hay un enfoque único *mejor* que funcione en todas las situaciones. Cada opción tiene sus fortalezas y debilidades.

HTML simple tiene la ventaja de ser simple. También se integra bien con texto. Tanto SVG como canvas te permiten dibujar texto, pero no te ayudarán a posicionar ese texto o envolverlo cuando ocupa más de

una línea. En una imagen basada en HTML, es mucho más fácil incluir bloques de texto.

SVG se puede utilizar para producir gráficos nítidos que se ven bien en cualquier nivel de zoom. A diferencia de HTML, está diseñado para dibujar y, por lo tanto, es más adecuado para ese propósito.

Tanto SVG como HTML construyen una estructura de datos (el DOM) que representa tu imagen. Esto hace posible modificar elementos después de ser dibujados. Si necesitas cambiar repetidamente una pequeña parte de una imagen grande en respuesta a lo que está haciendo el usuario o como parte de una animación, hacerlo en un canvas puede ser innecesariamente costoso. El DOM también nos permite registrar manipuladores de eventos de ratón en cada elemento de la imagen (incluso en formas dibujadas con SVG). No puedes hacer eso con canvas.

Pero el enfoque orientado a píxeles de canvas puede ser una ventaja al dibujar una gran cantidad de elementos pequeños. El hecho de que no construye una estructura de datos, sino que solo dibuja repetidamente sobre la misma superficie de píxeles, hace que canvas tenga un menor costo por forma.

También hay efectos, como renderizar una escena píxel por píxel (por ejemplo, usando un ray tracer) o procesar una imagen con JavaScript (desenfocarla o distorsionarla), que solo son prácticos con un elemento canvas.

En algunos casos, puede que desees combinar varias de estas técnicas. Por ejemplo, podrías dibujar un gráfico con SVG o canvas

pero mostrar información textual posicionando un elemento HTML encima de la imagen.

Para aplicaciones poco exigentes, realmente no importa mucho qué interfaz elijas. La visualización que construimos para nuestro juego en este capítulo podría haber sido implementada utilizando cualquiera de estas tres tecnologías gráficas ya que no necesita dibujar texto, manejar interacción del mouse o trabajar con una cantidad extraordinariamente grande de elementos.

RESUMEN

En este capítulo discutimos técnicas para dibujar gráficos en el navegador, centrándonos en el elemento `<canvas>`.

Un nodo canvas representa un área en un documento en la que nuestro programa puede dibujar. Este dibujo se realiza a través de un objeto de contexto de dibujo, creado con el método `getContext`.

La interfaz de dibujo 2D nos permite rellenar y trazar varias formas. La propiedad `fillStyle` del contexto determina cómo se rellenan las formas. Las propiedades `strokeStyle` y `lineWidth` controlan la forma en que se dibujan las líneas.

Los rectángulos y trozos de texto se pueden dibujar con una sola llamada a método. Los métodos `fillRect` y `strokeRect` dibujan rectángulos, y los métodos `fillText` y `strokeText` dibujan texto. Para crear formas personalizadas, primero debemos construir un camino.

Llamar a `beginPath` inicia un nuevo camino. Varios otros métodos agregan líneas y curvas al camino actual. Por ejemplo, `lineTo` puede agregar una línea recta. Cuando un camino está terminado, se puede rellenar con el método `fill` o trazarse con el método `stroke`.

Mover píxeles desde una imagen u otro canvas a nuestro canvas se hace con el método `drawImage`. Por defecto, este método dibuja toda la imagen fuente, pero al darle más parámetros, puedes copiar un área específica de la imagen. Utilizamos esto para nuestro juego copiando poses individuales del personaje del juego de una imagen que contenía muchas poses.

Las transformaciones te permiten dibujar una forma en múltiples orientaciones. Un contexto de dibujo 2D tiene una transformación actual que se puede cambiar con los métodos `translate`, `scale` y `rotate`. Estos afectarán todas las operaciones de dibujo subsiguientes. Un estado de transformación se puede guardar con el método `save` y restaurar con el método `restore`.

Al mostrar una animación en un canvas, se puede usar el método `clearRect` para borrar parte del canvas antes de volver a dibujarlo.

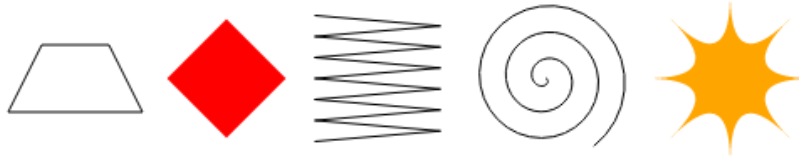
EJERCICIOS

FORMAS

Escribe un programa que dibuje las siguientes formas en un lienzo canvas:

1. Un trapecio (un rectángulo que es más ancho en un lado)

2. Un diamante rojo diamond (un rectángulo rotado 45 grados o $\frac{1}{4}\pi$ radianes)
3. Una línea en zigzag
4. Un espiral compuesta por 100 segmentos de línea recta
5. Una estrella amarilla star



Cuando dibujes las dos últimas, es posible que quieras consultar la explicación de `Math.cos` y `Math.sin` en [Capítulo 14](#), que describe cómo obtener coordenadas en un círculo utilizando estas funciones.

Recomiendo crear una función para cada forma. Pasa la posición y opcionalmente otras propiedades como el tamaño o el número de puntos, como parámetros. La alternativa, que es codificar números en todo tu código, tiende a hacer que el código sea innecesariamente difícil de leer y modificar.

EL GRÁFICO CIRCULAR

Anteriormente en este capítulo, vimos un programa de ejemplo que dibujaba un gráfico circular. Modifica este programa para que el nombre de cada categoría se muestre junto a la porción que la representa. Intenta encontrar una forma agradable de posicionar automáticamente este texto que funcione también para otros conjuntos de datos. Puedes asumir que las categorías son lo suficientemente grandes como para dejar espacio suficiente para sus etiquetas.

Podrías necesitar `Math.sin` y `Math.cos` de nuevo, que se describen en [Capítulo 14](#).

UNA PELOTA REBOTANDO

Utiliza la técnica de `requestAnimationFrame` que vimos en [Capítulo 14](#) y [Capítulo 16](#) para dibujar una caja con una pelota rebotando dentro. La pelota se mueve a una velocidad constante y rebota en los lados de la caja cuando los alcanza.

REFLEJO PRECALCULADO

Una desventaja de las transformaciones es que ralentizan el dibujo de mapas de bits. La posición y el tamaño de cada píxel deben ser transformados, y aunque es posible que los navegadores se vuelvan más inteligentes sobre las transformaciones en el futuro, actualmente causan un aumento medible en el tiempo que lleva dibujar un mapa de bits.

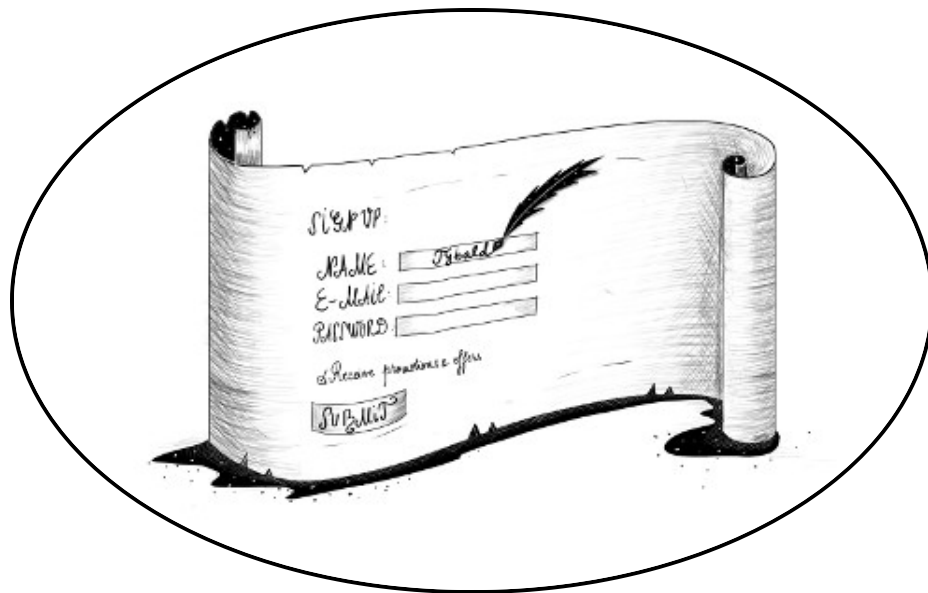
En un juego como el nuestro, en el que solo estamos dibujando un sprite transformado, esto no es un problema. Pero imagina que necesitamos dibujar cientos de personajes o miles de partículas giratorias de una explosión.

Piensa en una forma de permitirnos dibujar un personaje invertido sin cargar archivos de imagen adicionales y sin tener que hacer llamadas transformadas de `drawImage` en cada cuadro.

HTTP Y FORMULARIOS

“Lo que a menudo resultaba difícil para las personas entender sobre el diseño era que no había nada más allá de las URL, HTTP y HTML. No había una computadora central “controlando” la Web, no existía una sola red en la que funcionaran estos protocolos, ni siquiera una organización en algún lugar que “dirigiera” la Web. La Web no era una “cosa” física que existía en un cierto “lugar”. Era un “espacio” en el que la información podía existir.”

—Tim Berners-Lee



El *Protocolo de Transferencia de Hipertexto*, mencionado anteriormente en [Capítulo 13](#), es el mecanismo a través del cual se solicita y proporciona datos en la World Wide Web. Este capítulo describe el protocolo con más detalle y explica la forma en que JavaScript del navegador tiene acceso a él.

EL PROTOCOLO

Si escribes *eloquentjavascript.net/18_http.html* en la barra de direcciones de tu navegador, el navegador primero busca la dirección del servidor asociado con *eloquentjavascript.net* e intenta abrir una conexión TCP con él en el puerto 80, el puerto predeterminado para el tráfico HTTP. Si el servidor existe y acepta la conexión, el navegador podría enviar algo como esto:

```
GET /18_http.html HTTP/1.1
Host: eloquentjavascript.net
User-Agent: Nombre de tu navegador
```

Luego el servidor responde, a través de esa misma conexión.

```
HTTP/1.1 200 OK
Content-Length: 87320
Content-Type: text/html
Last-Modified: Vie, 13 Oct 2023 10:05:41 GMT

<!doctype html>
... el resto del documento
```

El navegador toma la parte de la respuesta después de la línea en blanco, su *cuerpo* (no confundir con la etiqueta HTML `<body>`), y lo muestra como un documento HTML.

La información enviada por el cliente se llama la *solicitud*. Comienza con esta línea:

```
GET /18_http.html HTTP/1.1
```

La primera palabra es el *método* de la solicitud. GET significa que queremos *obtener* el recurso especificado. Otros métodos comunes

son DELETE para eliminar un recurso, PUT para crearlo o reemplazarlo, y POST para enviar información a él. Cabe destacar que el servidor no está obligado a llevar a cabo cada solicitud que recibe. Si te acercas a un sitio web aleatorio y le dices que DELETE su página principal, probablemente se negará.

La parte después del nombre del método es la ruta del *recurso* al que aplica la solicitud. En el caso más simple, un recurso es simplemente un archivo en el servidor, pero el protocolo no lo requiere así. Un recurso puede ser cualquier cosa que pueda transferirse *como si fuera* un archivo. Muchos servidores generan las respuestas que producen al vuelo. Por ejemplo, si abres <https://github.com/marijnh>, el servidor buscará en su base de datos un usuario llamado “marijnh”, y si lo encuentra, generará una página de perfil para ese usuario. Después de la ruta del recurso, la primera línea de la solicitud menciona HTTP/1.1 para indicar la versión del protocolo HTTP que está utilizando.

En la práctica, muchos sitios utilizan la versión 2 de HTTP, que soporta los mismos conceptos que la versión 1.1 pero es mucho más complicada para que pueda ser más rápida. Los navegadores cambiarán automáticamente a la versión de protocolo adecuada al comunicarse con un servidor dado, y el resultado de una solicitud es el mismo independientemente de la versión utilizada. Dado que la versión 1.1 es más directa y más fácil de entender, la usaremos para ilustrar el protocolo.

La respuesta del servidor comenzará también con una versión, seguida del estado de la respuesta, primero como un código de estado de tres dígitos y luego como una cadena legible por humanos.

HTTP/1.1 200 OK

Los códigos de estado que comienzan con 2 indican que la solicitud tuvo éxito. Los códigos que comienzan con 4 significan que hubo un problema con la solicitud. El 404 es probablemente el código de estado de HTTP más famoso, lo que significa que el recurso no se pudo encontrar. Los códigos que comienzan con 5 indican que ocurrió un error en el servidor y la solicitud no es la responsable.

La primera línea de una solicitud o respuesta puede ir seguida de cualquier número de *cabeceras*. Estas son líneas en la forma `nombre: valor` que especifican información adicional sobre la solicitud o respuesta. Estas cabeceras eran parte del ejemplo de respuesta:

```
Content-Length: 87320
Content-Type: text/html
Last-Modified: Fri, 13 Oct 2023 10:05:41 GMT
```

Esto nos indica el tamaño y tipo del documento de respuesta. En este caso, es un documento HTML de 87,320 bytes. También nos dice cuándo se modificó por última vez ese documento.

El cliente y servidor son libres de decidir qué cabeceras incluir en sus solicitudes o respuestas. Sin embargo, algunas de ellas son necesarias para que todo funcione. Por ejemplo, sin la cabecera `Content-Type` en la respuesta, el navegador no sabrá cómo mostrar el documento.

Después de las cabeceras, tanto las solicitudes como las respuestas pueden incluir una línea en blanco seguida de un cuerpo, que contiene el documento real que se envía. Las solicitudes GET y

DELETE no envían ningún dato, pero las solicitudes PUT y POST sí lo hacen. Algunos tipos de respuestas, como las respuestas de error, tampoco requieren un cuerpo.

NAVEGADORES Y HTTP

Como vimos, un navegador hará una solicitud cuando introducimos una URL en la barra de direcciones. Cuando la página HTML resultante hace referencia a otros archivos, como imágenes y archivos de JavaScript, el navegador los recuperará también.

Un sitio web moderadamente complicado puede incluir fácilmente entre 10 y 200 recursos. Para poder obtenerlos rápidamente, los navegadores harán varias solicitudes GET simultáneamente en lugar de esperar las respuestas una por una. Las páginas HTML pueden incluir formularios, que permiten al usuario completar información y enviarla al servidor. A continuación se muestra un ejemplo de un formulario:

```
<form method="GET" action="example/message.html">
  <p>Nombre: <input type="text" name="name"></p>
  <p>Mensaje:<br><textarea name="message"></textarea></p>
  <p><button type="submit">Enviar</button></p>
</form>
```

Este código describe un formulario con dos campos: uno pequeño que pide un nombre y otro más grande para escribir un mensaje. Cuando se hace clic en el botón Enviar, el formulario se envía, lo que significa que el contenido de sus campos se empaqueta en una solicitud HTTP y el navegador navega hacia el resultado de esa solicitud.

Cuando el atributo `method` del elemento `<form>` es `GET` (o se omite), la información del formulario se agrega al final de la URL de `action` como una cadena de consulta. El navegador podría hacer una solicitud a esta URL:

```
GET /example/message.html?name=Jean&message=Yes%3F HTTP/1.1
```

El signo de interrogación indica el final de la parte de la ruta de la URL y el inicio de la consulta. Le siguen pares de nombres y valores, correspondientes al atributo `name` en los elementos del campo del formulario y al contenido de esos elementos, respectivamente. Un carácter ampersand (`&`) se utiliza para separar los pares.

El mensaje real codificado en la URL es “Yes?”, pero el signo de interrogación se reemplaza por un código extraño. Algunos caracteres en las cadenas de consulta deben ser escapados. El signo de interrogación, representado como `%3F`, es uno de ellos. Parece haber una regla no escrita de que cada formato necesita su propia forma de escapar caracteres. Este, llamado codificación de URL, utiliza un signo de porcentaje seguido de dos dígitos hexadecimales (base 16) que codifican el código de caracteres. En este caso, `3F`, que es 63 en notación decimal, es el código de un signo de interrogación. JavaScript proporciona las funciones `encodeURIComponent` y `decodeURIComponent` para codificar y decodificar este formato.

```
console.log(encodeURIComponent("Yes?"));  
// → Yes%3F  
console.log(decodeURIComponent("Yes%3F"));  
// → Yes?
```

Si cambiamos el atributo `method` del formulario HTML en el ejemplo que vimos anteriormente a `POST`, la solicitud HTTP

realizada para enviar el formulario utilizará el método POST y colocará la cadena de consulta en el cuerpo de la solicitud, en lugar de agregarla a la URL.

```
POST /example/message.html HTTP/1.1
Content-length: 24
Content-type: application/x-www-form-urlencoded

name=Jean&message=Yes%3F
```

Las solicitudes GET deben utilizarse para solicitudes que no tengan efectos secundarios, sino simplemente para solicitar información. Las solicitudes que cambian algo en el servidor, como por ejemplo crear una nueva cuenta o publicar un mensaje, deben expresarse con otros métodos, como POST. El software del lado del cliente, como un navegador, sabe que no debe hacer solicitudes POST a ciegas, pero a menudo implícitamente realiza solicitudes GET, por ejemplo, para precargar un recurso que cree que pronto el usuario necesitará. Volveremos a hablar de formularios y cómo interactuar con ellos desde JavaScript [más adelante en el capítulo](#).

FETCH

La interfaz a través de la cual JavaScript del navegador puede hacer solicitudes HTTP se llama `fetch`.

```
fetch("ejemplo/datos.txt").then(response => {
  console.log(response.status);
  // → 200
  console.log(response.headers.get("Content-Type"));
  // → text/plain
});
```


Llamar a `fetch` devuelve una promesa que se resuelve en un objeto `Response` que contiene información sobre la respuesta del servidor, como su código de estado y sus encabezados. Los encabezados están envueltos en un objeto similar a un `Map` que trata sus claves (los nombres de los encabezados) como insensibles a mayúsculas y minúsculas porque los nombres de los encabezados no deben ser sensibles a mayúsculas y minúsculas. Esto significa que `headers.get("Content-Type")` y `headers.get("content-TYPE")` devolverán el mismo valor.

Ten en cuenta que la promesa devuelta por `fetch` se resuelve con éxito incluso si el servidor responde con un código de error. También puede ser rechazada si hay un error de red o si el servidor al que se dirige la solicitud no se puede encontrar.

El primer argumento de `fetch` es la URL que se debe solicitar. Cuando esa URL no comienza con un nombre de protocolo (como *http:*), se trata como *relativa*, lo que significa que se interpreta en relación con el documento actual. Cuando comienza con una barra (/), reemplaza la ruta actual, que es la parte después del nombre del servidor. Cuando no lo hace, la parte de la ruta actual hasta e incluyendo su último carácter de barra se coloca al principio de la URL relativa.

Para acceder al contenido real de una respuesta, puedes usar su método `text`. Debido a que la promesa inicial se resuelve tan pronto como se han recibido los encabezados de la respuesta y porque leer el cuerpo de la respuesta podría llevar un poco más de tiempo, esto devuelve nuevamente una promesa.

```
fetch("ejemplo/datos.txt")
  .then(resp => resp.text())
  .then(text => console.log(text));
// → Este es el contenido de datos.txt
```

Un método similar, llamado `json`, devuelve una promesa que se resuelve al valor que obtienes al analizar el cuerpo como JSON o se rechaza si no es un JSON válido.

Por defecto, `fetch` utiliza el método GET para realizar su solicitud y no incluye un cuerpo de solicitud. Puedes configurarlo de manera diferente pasando un objeto con opciones adicionales como segundo argumento. Por ejemplo, esta solicitud intenta eliminar `ejemplo/datos.txt`:

```
fetch("ejemplo/datos.txt", {method: "DELETE"}).then(resp => {
  console.log(resp.status);
  // → 405
});
```

El código de estado 405 significa “método no permitido”, la forma en que un servidor HTTP dice “Me temo que no puedo hacer eso”.

Para agregar un cuerpo de solicitud, puedes incluir una opción `body`. Para establecer cabeceras, está la opción `headers`. Por ejemplo, esta solicitud incluye una cabecera `Range`, que indica al servidor que devuelva solo una parte de un documento.

```
fetch("ejemplo/datos.txt", {headers: {Range: "bytes=8-19"}})
  .then(resp => resp.text())
  .then(console.log);
// → el contenido
```

El navegador automáticamente añadirá algunas cabeceras de solicitud, como “Host” y aquellas necesarias para que el servidor pueda determinar el tamaño del cuerpo. Sin embargo, añadir tus propias cabeceras es muchas veces útil para incluir cosas como información de autenticación o para indicar al servidor en qué formato de archivo te gustaría recibir.

AISLAMIENTO HTTP

Realizar solicitudes HTTP en scripts de páginas web plantea nuevamente preocupaciones sobre seguridad. La persona que controla el script puede no tener los mismos intereses que la persona en cuya computadora se está ejecutando. Específicamente, si visito *themafia.org*, no quiero que sus scripts puedan hacer una solicitud a *mybank.com*, utilizando información de identificación de mi navegador, con instrucciones para transferir todo mi dinero.

Por esta razón, los navegadores nos protegen al impedir que los scripts hagan solicitudes HTTP a otros dominios (nombres como *themafia.org* y *mybank.com*).

Esto puede ser un problema molesto al construir sistemas que necesitan acceder a varios dominios por razones legítimas. Afortunadamente, los servidores pueden incluir una cabecera como esta en sus respuestas para indicar explícitamente al navegador que está bien que la solicitud provenga de otro dominio:

```
Access-Control-Allow-Origin: *
```

APRECIANDO HTTP

Cuando se construye un sistema que requiere comunicación entre un programa JavaScript que se ejecuta en el navegador (lado del cliente) y un programa en un servidor (lado del servidor), hay varias formas diferentes de modelar esta comunicación.

Un modelo comúnmente utilizado es el de las *llamadas de procedimiento remoto*. En este modelo, la comunicación sigue los patrones de llamadas de función normales, excepto que la función en realidad se está ejecutando en otra máquina. Llamarla implica hacer una solicitud al servidor que incluye el nombre de la función y sus argumentos. La respuesta a esa solicitud contiene el valor devuelto.

Cuando se piensa en términos de llamadas de procedimiento remoto, HTTP es simplemente un vehículo de comunicación, y es muy probable que escribas una capa de abstracción que lo oculte por completo.

Otro enfoque es construir tu comunicación en torno al concepto de recursos y métodos HTTP. En lugar de un procedimiento remoto llamado `addUser`, usas una solicitud PUT a `/usuarios/larry`. En lugar de codificar las propiedades de ese usuario en argumentos de función, defines un formato de documento JSON (o utilizas un formato existente) que represente a un usuario. El cuerpo de la solicitud PUT para crear un nuevo recurso es entonces dicho documento. Se obtiene un recurso realizando una solicitud GET a la URL del recurso (por ejemplo, `/usuario/larry`), que de nuevo devuelve el documento que representa al recurso. Este segundo enfoque facilita el uso de algunas de las características que proporciona HTTP, como el soporte para la caché de recursos (mantener una copia de un recurso en el cliente para un acceso

rápido). Los conceptos utilizados en HTTP, que están bien diseñados, pueden proporcionar un conjunto útil de principios para diseñar la interfaz de tu servidor.

SEGURIDAD Y HTTPS

Los datos que viajan por Internet tienden a seguir un largo y peligroso camino. Para llegar a su destino, deben pasar por cualquier cosa, desde puntos de acceso Wi-Fi de cafeterías hasta redes controladas por varias empresas y estados. En cualquier punto a lo largo de su ruta, pueden ser inspeccionados o incluso modificados.

Si es importante que algo se mantenga en secreto, como la contraseña de tu cuenta de correo electrónico, o que llegue a su destino sin modificaciones, como el número de cuenta al que transfieres dinero a través del sitio web de tu banco, HTTP simple no es suficiente.

El protocolo seguro HTTP, utilizado para URLs que comienzan con *https://*, envuelve el tráfico HTTP de una manera que dificulta su lectura y manipulación. Antes de intercambiar datos, el cliente verifica que el servidor sea quien dice ser, solicitándole que demuestre que tiene un certificado criptográfico emitido por una autoridad de certificación que el navegador reconoce. Luego, todos los datos que pasan por la conexión están encriptados de una manera que debería evitar el espionaje y la manipulación.

Así, cuando funciona correctamente, HTTPS evita que otras personas se hagan pasar por el sitio web con el que estás intentando comunicarte *y* que espíen tu comunicación. No es perfecto, y ha habido varios incidentes en los que HTTPS falló debido a certificados

falsificados o robados y software defectuoso, pero es *mucho* más seguro que el simple HTTP.

CAMPOS DE FORMULARIO

Los formularios fueron diseñados originalmente para la Web pre-JavaScript para permitir que los sitios web envíen información enviada por el usuario en una solicitud HTTP. Este diseño asume que la interacción con el servidor siempre ocurre navegando a una nueva página.

Pero sus elementos son parte del DOM al igual que el resto de la página, y los elementos DOM que representan los campos de formulario admiten una serie de propiedades y eventos que no están presentes en otros elementos. Esto hace posible inspeccionar y controlar dichos campos de entrada con programas JavaScript y hacer cosas como agregar nueva funcionalidad a un formulario o utilizar formularios y campos como bloques de construcción en una aplicación JavaScript.

Un formulario web consiste en cualquier número de campos de entrada agrupados en una etiqueta `<form>`. HTML permite varios estilos diferentes de campos, que van desde simples casillas de verificación de encendido/apagado hasta menús desplegables y campos para entrada de texto. Este libro no intentará discutir exhaustivamente todos los tipos de campos, pero comenzaremos con una vista general aproximada.

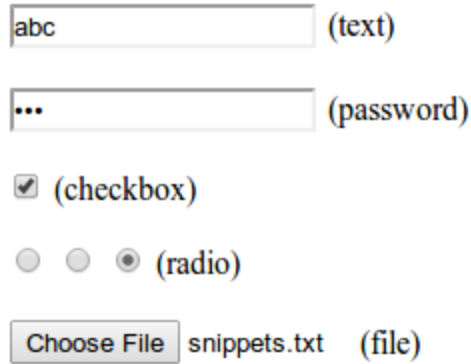
Muchos tipos de campos utilizan la etiqueta `<input>`. El atributo `type` de esta etiqueta se utiliza para seleccionar el estilo del campo. Estos son algunos tipos comúnmente utilizados de `<input>`:

texto	Un campo de una línea campo de texto
contraseña	Igual que texto pero oculta el texto que se escribe
casilla de verificación	Un interruptor de encendido/apagado
color	Un color
fecha	Una fecha de calendario
radio	(Parte de) un campo de opción múltiple
archivo	Permite al usuario elegir un archivo de su computadora

Los campos de formulario no necesariamente tienen que aparecer en una etiqueta `<form>`. Puedes ponerlos en cualquier parte de una página. Campos sin formulario no pueden ser enviados (solo un formulario en su totalidad puede), pero al responder a la entrada con JavaScript, a menudo no queremos enviar nuestros campos de forma normal de todos modos.

```
<p><input type="texto" value="abc"> (texto)</p>
<p><input type="contraseña" value="abc"> (contraseña)</p>
<p><input type="casilla de verificación" checked> (casilla de
verificación)</p>
<p><input type="color" value="naranja"> (color)</p>
<p><input type="fecha" value="2023-10-13"> (fecha)</p>
<p><input type="radio" value="A" name="elección">
  <input type="radio" value="B" name="elección" checked>
  <input type="radio" value="C" name="elección"> (radio)</p>
<p><input type="archivo"> (archivo)</p>
```

Los campos creados con este código HTML lucen así:



abc (text)

... (password)

☒ (checkbox)

☐ ☐ ☒ (radio)

Choose File snippets.txt (file)

La interfaz de JavaScript para estos elementos difiere según el tipo de elemento.

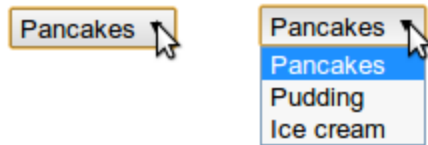
Los campos de texto de varias líneas tienen su propia etiqueta, `<textarea>`, principalmente porque sería incómodo utilizar un atributo para especificar un valor de inicio de varias líneas. La etiqueta `<textarea>` requiere una etiqueta de cierre `</textarea>` coincidente y utiliza el texto entre esas dos etiquetas, en lugar del atributo `valor`, como texto de inicio.

```
<textarea>
uno
dos
tres
</textarea>
```

Finalmente, la etiqueta `<select>` se usa para crear un campo que permite al usuario seleccionar de varias opciones predefinidas.

```
<select>
  <option>Tortitas</option>
  <option>Pudín</option>
  <option>Helado</option>
</select>
```

Dicho campo luce así:



Cada vez que cambia el valor de un campo de formulario, se desencadenará un evento "cambio".

ENFOQUE

A diferencia de la mayoría de elementos en documentos HTML, los campos de formulario pueden obtener *enfoque de teclado*. Cuando se hace clic, se mueve con la tecla TAB, o se activa de alguna otra manera, se convierten en el elemento activo actual y en el receptor de la entrada de teclado.

Por lo tanto, puedes escribir en un campo de texto solo cuando está enfocado. Otros campos responden diferentemente a los eventos de teclado. Por ejemplo, un menú `<select>` intenta moverse a la opción que contiene el texto que el usuario escribió y responde a las teclas de flecha moviendo su selección hacia arriba y hacia abajo.

Podemos controlar el focus desde JavaScript con los métodos `focus` y `blur`. El primero mueve el enfoque al elemento del DOM en el que se llama, y el segundo elimina el enfoque. El valor en `document.activeElement` corresponde al elemento actualmente enfocado.

```
<input type="text">
<script>
  document.querySelector("input").focus();
  console.log(document.activeElement.tagName);
  // → INPUT
  document.querySelector("input").blur();
  console.log(document.activeElement.tagName);
```

```
// → BODY  
</script>
```

Para algunas páginas, se espera que el usuario desee interactuar inmediatamente con un campo de formulario. JavaScript se puede utilizar para enfocar este campo cuando se carga el documento, pero HTML también proporciona el atributo `autofocus`, que produce el mismo efecto al mismo tiempo que le indica al navegador lo que estamos tratando de lograr. Esto le da al navegador la opción de deshabilitar el comportamiento cuando no es apropiado, como cuando el usuario ha puesto el enfoque en otra parte.

Los navegadores permiten al usuario mover el enfoque a través del documento presionando la tecla `TAB` para pasar al siguiente elemento enfocable, y `SHIFT-TAB` para retroceder al elemento anterior. Por defecto, los elementos se visitan en el orden en que aparecen en el documento. Es posible usar el atributo `tabindex` para cambiar este orden. El siguiente ejemplo de documento permitirá que el enfoque salte del campo de texto al botón OK, en lugar de pasar primero por el enlace de ayuda:

```
<input type="text" tabindex=1> <a href=".">(ayuda)</a>  
<button onclick="console.log('ok')" tabindex=2>OK</button>
```

Por defecto, la mayoría de los tipos de elementos HTML no pueden ser enfocados. Pero se puede agregar un atributo `tabindex` a cualquier elemento para hacerlo enfocable. Un `tabindex` de 0 hace que un elemento sea enfocable sin afectar el orden de enfoque.

CAMPOS DESHABILITADOS

Todos los campos de formulario pueden ser *deshabilitados* a través de su atributo `disabled`. Es un atributo que se puede especificar sin valor; el simple hecho de que esté presente deshabilita el elemento.

```
<button>Estoy bien</button>  
<button disabled>Estoy fuera</button>
```

Los campos deshabilitados no pueden ser enfocados ni modificados, y los navegadores los muestran de color gris y atenuados.



Cuando un programa está en proceso de manejar una acción provocada por algún botón u otro control que podría requerir comunicación con el servidor y por lo tanto llevar un tiempo, puede ser una buena idea deshabilitar el control hasta que la acción haya terminado. De esta forma, cuando el usuario se impacienta y hace clic nuevamente, no repiten accidentalmente su acción.

EL FORMULARIO EN SU TOTALIDAD

Cuando un field está contenido en un elemento `<form>`, su elemento DOM tendrá una propiedad `form` que enlaza de vuelta al elemento DOM del formulario. El elemento `<form>`, a su vez, tiene una propiedad llamada `elements` que contiene una colección similar a un array de los campos dentro de él.

El atributo `name` de un campo de formulario determina la forma en que se identificará su valor cuando se submitee el formulario. También se puede utilizar como nombre de propiedad al acceder a la

propiedad `elements` del formulario, la cual actúa tanto como un objeto similar a un array (accesible por número) como un mapa (accesible por nombre).

```
<form action="ejemplo/enviar.html">
  Nombre: <input type="text" name="nombre"><br>
  Contraseña: <input type="password" name="contraseña"><br>
  <button type="submit">Ingresar</button>
</form>
<script>
  let formulario = document.querySelector("form");
  console.log(formulario.elements[1].type);
  // → password
  console.log(formulario.elements.contraseña.type);
  // → password
  console.log(formulario.elements.nombre.form == formulario);
  // → true
</script>
```

Un botón con un atributo `type` de `submit` hará que, al presionarlo, se submita el formulario. Presionar `ENTER` cuando un campo de formulario está enfocado tendrá el mismo efecto.

Enviar un formulario normalmente significa que el navegador se dirige a la página indicada por el atributo `action` del formulario, utilizando ya sea una solicitud `GET` o `POST`. Pero antes de que eso ocurra, se dispara un evento `"submit"`. Puedes manejar este evento con JavaScript y prevenir este comportamiento por defecto llamando a `preventDefault` en el objeto de evento.

```
<form>
  Valor: <input type="text" name="valor">
  <button type="submit">Guardar</button>
</form>
<script>
  let formulario = document.querySelector("form");
```

```
formulario.addEventListener("submit", evento => {  
    console.log("Guardando valor",  
formulario.elements.valor.value);  
    evento.preventDefault();  
});  
</script>
```

Interceptar los eventos "submit" en JavaScript tiene varios usos. Podemos escribir código para verificar que los valores ingresados por el usuario tengan sentido y mostrar inmediatamente un mensaje de error en lugar de enviar el formulario. O podemos deshabilitar completamente la forma regular de enviar el formulario, como en el ejemplo, y hacer que nuestro programa maneje la entrada, posiblemente utilizando `fetch` para enviarla a un servidor sin recargar la página.

CAMPOS DE TEXTO

Los campos creados por etiquetas `<textarea>`, o etiquetas `<input>` con un tipo de `text` o `password`, comparten una interfaz común. Sus elementos DOM tienen una propiedad `value` que contiene su contenido actual como un valor de cadena. Establecer esta propiedad a otra cadena cambia el contenido del campo.

Las propiedades `selectionStart` y `selectionEnd` de los campos de texto nos brindan información sobre la posición del cursor y la selección en el texto. Cuando no se ha seleccionado nada, estas dos propiedades contienen el mismo número, indicando la posición del cursor. Por ejemplo, 0 indica el inicio del texto, y 10 indica que el cursor está después del 10^o carácter. Cuando se selecciona parte del campo, las dos propiedades serán diferentes,

dándonos el inicio y el final del texto seleccionado. Al igual que `value`, estas propiedades también se pueden escribir.

Imagina que estás escribiendo un artículo sobre Khasekhemwy pero tienes problemas para deletrear su nombre. El siguiente código vincula una etiqueta `<textarea>` con un controlador de eventos que, al presionar F2, inserta la cadena “Khasekhemwy” por ti.

```
<textarea></textarea>
<script>
  let textarea = document.querySelector("textarea");
  textarea.addEventListener("keydown", event => {
    if (event.key == "F2") {
      replaceSelection(textarea, "Khasekhemwy");
      event.preventDefault();
    }
  });
  function replaceSelection(field, word) {
    let from = field.selectionStart, to = field.selectionEnd;
    field.value = field.value.slice(0, from) + word +
      field.value.slice(to);
    // Coloca el cursor después de la palabra
    field.selectionStart = from + word.length;
    field.selectionEnd = from + word.length;
  }
</script>
```

La función `replaceSelection` reemplaza la parte actualmente seleccionada del contenido de un campo de texto con la palabra proporcionada y luego mueve el cursor después de esa palabra para que el usuario pueda continuar escribiendo.

El evento `"change"` para un campo de texto no se activa cada vez que se escribe algo. En cambio, se activa cuando el campo pierde el enfoque después de que su contenido haya cambiado. Para responder de inmediato a los cambios en un campo de texto, se debe

registrar un controlador para el evento "input", que se activa cada vez que el usuario escribe un carácter, elimina texto o de otra manera manipula el contenido del campo.

El siguiente ejemplo muestra un campo de texto y un contador que muestra la longitud actual del texto en el campo:

```
<input type="text"> longitud: <span id="length">0</span>
<script>
  let texto = document.querySelector("input");
  let output = document.querySelector("#length");
  texto.addEventListener("input", () => {
    output.textContent = texto.value.length;
  });
</script>
```

CASILLAS DE VERIFICACIÓN Y BOTONES DE RADIO

Un campo de casilla de verificación es un interruptor binario. Su valor se puede extraer o cambiar a través de su propiedad `checked`, que contiene un valor Booleano.

```
<label>
  <input type="checkbox" id="purple"> Hacer esta página morada
</label>
<script>
  let casillaVerificacion = document.querySelector("#purple");
  casillaVerificacion.addEventListener("change", () => {
    document.body.style.background =
      casillaVerificacion.checked ? "mediumpurple" : "";
  });
</script>
```

La etiqueta `<label>` asocia un fragmento de documento con un campo de entrada. Hacer clic en cualquier parte de la etiqueta

activará el campo, lo enfocará e invertirá su valor cuando sea un casilla de verificación o un botón de radio.

Un botón de radio es similar a una casilla de verificación, pero está vinculado implícitamente a otros botones de radio con el mismo atributo name para que solo uno de ellos pueda estar activo en cualquier momento.

```
Color:
<label>
  <input type="radio" name="color" value="orange"> Naranja
</label>
<label>
  <input type="radio" name="color" value="lightgreen"> Verde claro
</label>
<label>
  <input type="radio" name="color" value="lightblue"> Azul claro
</label>
<script>
  let buttons = document.querySelectorAll("[name=color]");
  for (let button of Array.from(buttons)) {
    button.addEventListener("change", () => {
      document.body.style.background = button.value;
    });
  }
</script>
```

Los corchetes cuadrados en la consulta CSS proporcionada a `querySelectorAll` se utilizan para hacer coincidir atributos. Selecciona elementos cuyo atributo name es "color".

CAMPOS DE SELECCIÓN

Los campos de selección son conceptualmente similares a los botones de radio, ya que también permiten al usuario elegir entre un conjunto de opciones. Sin embargo, mientras que un botón de radio

pone el diseño de las opciones bajo nuestro control, la apariencia de una etiqueta `<select>` está determinada por el navegador.

Los campos de selección también tienen una variante que se asemeja más a una lista de casillas de verificación que a botones de radio. Cuando se le otorga el atributo `multiple`, una etiqueta `<select>` permitirá al usuario seleccionar cualquier número de opciones, en lugar de una sola opción. Mientras que un campo de selección regular se muestra como un control de *lista desplegable*, que muestra las opciones inactivas solo cuando lo abres, un campo con `multiple` habilitado muestra múltiples opciones al mismo tiempo, permitiendo al usuario habilitar o deshabilitarlas individualmente.

Cada etiqueta `<option>` tiene un valor. Este valor se puede definir con un atributo `value`. Cuando este no se proporciona, el texto dentro de la opción se considerará como su valor. La propiedad `value` de un elemento `<select>` refleja la opción actualmente seleccionada. Sin embargo, para un campo `multiple`, esta propiedad no significa mucho, ya que dará el valor de solo *una* de las opciones actualmente seleccionadas.

Las etiquetas `<option>` para un campo `<select>` pueden ser accedidas como un objeto similar a un array a través de la propiedad `options` del campo. Cada opción tiene una propiedad llamada `selected`, que indica si esa opción está actualmente seleccionada. La propiedad también se puede escribir para seleccionar o deseleccionar una opción.

Este ejemplo extrae los valores seleccionados de un campo de selección `multiple` y los utiliza para componer un número binario

a partir de bits individuales. Mantén pulsado CONTROL (o COMMAND en un Mac) para seleccionar múltiples opciones.

```
<select multiple>
  <option value="1">0001</option>
  <option value="2">0010</option>
  <option value="4">0100</option>
  <option value="8">1000</option>
</select> = <span id="output">0</span>
<script>
  let select = document.querySelector("select");
  let output = document.querySelector("#output");
  select.addEventListener("change", () => {
    let number = 0;
    for (let option of Array.from(select.options)) {
      if (option.selected) {
        number += Number(option.value);
      }
    }
    output.textContent = number;
  });
</script>
```

CAMPOS DE ARCHIVO

Los campos de archivo fueron diseñados originalmente como una forma de subir archivos desde la máquina del usuario a través de un formulario. En los navegadores modernos, también proporcionan una forma de leer dichos archivos desde programas JavaScript. El campo actúa como una especie de guardián. El script no puede simplemente comenzar a leer archivos privados desde la computadora del usuario, pero si el usuario selecciona un archivo en dicho campo, el navegador interpreta esa acción como que el script puede leer el archivo.

Un campo de archivo suele parecerse a un botón etiquetado con algo como “elegir archivo” o “explorar”, con información sobre el archivo elegido al lado.

```
<input type="file">
<script>
  let input = document.querySelector("input");
  input.addEventListener("change", () => {
    if (input.files.length > 0) {
      let file = input.files[0];
      console.log("Has elegido", file.name);
      if (file.type) console.log("Tiene tipo", file.type);
    }
  });
</script>
```

La propiedad `files` de un elemento campo de archivo es un objeto similar a un arreglo (una vez más, no es un arreglo real) que contiene los archivos elegidos en el campo. Inicialmente está vacío. La razón por la que no hay simplemente una propiedad `file` es que los campos de archivo también admiten un atributo `multiple`, lo que permite seleccionar varios archivos al mismo tiempo.

Los objetos en `files` tienen propiedades como `name` (el nombre de archivo), `size` (el tamaño del archivo en bytes, que son trozos de 8 bits) y `type` (el tipo de medio del archivo, como `text/plain` o `image/jpeg`).

Lo que no tiene es una propiedad que contenga el contenido del archivo. Acceder a eso es un poco más complicado. Dado que leer un archivo desde el disco puede llevar tiempo, la interfaz es asíncrona para evitar que se congele la ventana.

```

<input type="file" multiple>
<script>
  let input = document.querySelector("input");
  input.addEventListener("change", () => {
    for (let file of Array.from(input.files)) {
      let reader = new FileReader();
      reader.addEventListener("load", () => {
        console.log("El archivo", file.name, "comienza con",
          reader.result.slice(0, 20));
      });
      reader.readAsText(file);
    }
  });
</script>

```

La lectura de un archivo se realiza creando un objeto `FileReader`, registrando un controlador de eventos `"load"` para él y llamando a su método `readAsText`, dándole el archivo que queremos leer. Una vez que la carga finaliza, la propiedad `result` del lector contiene el contenido del archivo.

Los `FileReaders` también disparan un evento `"error"` cuando la lectura del archivo falla por cualquier motivo. El objeto de error en sí terminará en la propiedad `error` del lector. Esta interfaz fue diseñada antes de que las promesas se convirtieran en parte del lenguaje. Podrías envolverlo en una promesa de la siguiente manera:

```

function readFileText(file) {
  return new Promise((resolve, reject) => {
    let reader = new FileReader();
    reader.addEventListener(
      "load", () => resolve(reader.result));
    reader.addEventListener(
      "error", () => reject(reader.error));
    reader.readAsText(file);
  });
}

```

ALMACENANDO DATOS DEL LADO DEL CLIENTE

Páginas simples de HTML con un poco de JavaScript pueden ser un gran formato para “mini aplicaciones” - pequeños programas auxiliares que automatizan tareas básicas. Conectando unos cuantos campos de formulario con controladores de eventos, puedes hacer desde convertir entre centímetros y pulgadas hasta calcular contraseñas a partir de una contraseña maestra y un nombre de sitio web.

Cuando una aplicación así necesita recordar algo entre sesiones, no puedes usar las vinculaciones de JavaScript, ya que estas se descartan cada vez que se cierra la página. Podrías configurar un servidor, conectarlo a Internet y hacer que tu aplicación almacene algo allí. Veremos cómo hacerlo en el [Capítulo 20](#). Pero eso implica mucho trabajo extra y complejidad. A veces es suficiente con mantener los datos en el navegador.

El objeto `localStorage` se puede utilizar para almacenar datos de una manera que sobreviva a las recargas de página. Este objeto te permite guardar valores de cadena bajo nombres.

```
localStorage.setItem("nombre de usuario", "marijn");  
console.log(localStorage.getItem("nombre de usuario"));  
// → marijn  
localStorage.removeItem("nombre de usuario");
```

Un valor en `localStorage` permanece hasta que se sobrescribe, se elimina con `removeItem` o el usuario elimina sus datos locales.

Los sitios de diferentes dominios obtienen compartimentos de almacenamiento diferentes. Eso significa que los datos almacenados

en local Storage por un sitio web dado, en principio, solo pueden ser leídos (y sobrescritos) por scripts en ese mismo sitio.

Los navegadores aplican un límite en el tamaño de los datos que un sitio puede almacenar en local Storage. Esta restricción, junto con el hecho de que llenar los discos duros de la gente con basura no es realmente rentable, evita que la función ocupe demasiado espacio.

El siguiente código implementa una aplicación rudimentaria de toma de notas. Mantiene un conjunto de notas con nombres y permite al usuario editar notas y crear nuevas.

```
Notas: <select></select> <button>Añadir</button><br>
<textarea style="width: 100%"></textarea>

<script>
  let list = document.querySelector("select");
  let note = document.querySelector("textarea");

  let state;
  function setState(nuevoEstado) {
    list.textContent = "";
    for (let nombre of Object.keys(nuevoEstado.notes)) {
      let option = document.createElement("option");
      option.textContent = nombre;
      if (nuevoEstado.selected == nombre) option.selected = true;
      list.appendChild(option);
    }
    note.value = nuevoEstado.notes[nuevoEstado.selected];

    localStorage.setItem("Notas", JSON.stringify(nuevoEstado));
    state = nuevoEstado;
  }
  setState(JSON.parse(localStorage.getItem("Notas"))) ?? {
    notes: {"lista de compras": "Zanahorias\nPasas"},
    selected: "lista de compras"
  });
});
```

```

list.addEventListener("change", () => {
  setState({notes: state.notes, selected: list.value});
});
note.addEventListener("change", () => {
  let {selected} = state;
  setState({
    notes: {...state.notes, [selected]: note.value},
    selected
  });
});
document.querySelector("button")
  .addEventListener("click", () => {
    let nombre = prompt("Nombre de la nota");
    if (nombre) setState({
      notes: {...state.notes, [nombre]: ""},
      selected: nombre
    });
  });
</script>

```

El script obtiene su estado inicial del valor "Notas" almacenado en `localStorage` o, si está ausente, crea un estado de ejemplo que solo contiene una lista de compras. Leer un campo que no existe en `localStorage` devolverá `null`. Pasar `null` a `JSON.parse` hará que analice la cadena "`null`" y devuelva `null`. Por tanto, en una situación como esta se puede utilizar el operador `??` para proporcionar un valor predeterminado.

El método `setState` se asegura de que el DOM muestre un estado dado y almacena el nuevo estado en `localStorage`. Los controladores de eventos llaman a esta función para moverse a un nuevo estado.

La sintaxis `...` en el ejemplo se utiliza para crear un nuevo objeto que es un clon del antiguo `state.notes`, pero con una propiedad añadida o sobrescrita. Utiliza la sintaxis `spread` para primero añadir

las propiedades del objeto antiguo y luego establecer una nueva propiedad. La notación de corchetes cuadrados en el literal del objeto se utiliza para crear una propiedad cuyo nombre se basa en algún valor dinámico.

Existe otro objeto, similar a `localStorage`, llamado `sessionStorage`. La diferencia entre los dos es que el contenido de `sessionStorage` se olvida al final de cada *sesión*, lo que en la mayoría de los navegadores significa cada vez que se cierra el navegador.

RESUMEN

En este capítulo, discutimos cómo funciona el protocolo HTTP. Un *cliente* envía una solicitud, que contiene un método (generalmente GET) y una ruta que identifica un recurso. El *servidor* luego decide qué hacer con la solicitud y responde con un código de estado y un cuerpo de respuesta. Tanto las solicitudes como las respuestas pueden contener encabezados que proporcionan información adicional. La interfaz a través de la cual JavaScript del navegador puede realizar solicitudes HTTP se llama `fetch`. Realizar una solicitud se ve así:

```
fetch("/18_http.html").then(r => r.text()).then(text => {  
  console.log(`La página comienza con ${text.slice(0, 15)}`);  
});
```

Los navegadores hacen solicitudes GET para obtener los recursos necesarios para mostrar una página web. Una página también puede contener formularios, que permiten enviar información ingresada

por el usuario como una solicitud de una nueva página cuando se envía el formulario.

HTML puede representar varios tipos de campos de formulario, como campos de texto, casillas de verificación, campos de selección múltiple y selectores de archivos.

Estos campos pueden ser inspeccionados y manipulados con JavaScript. Disparan el evento "change" al cambiar, disparan el evento "input" al escribir texto y reciben eventos del teclado cuando tienen el foco del teclado. Propiedades como `value` (para campos de texto y select) o `checked` (para casillas de verificación y botones de radio) se utilizan para leer o establecer el contenido del campo.

Cuando un formulario se envía, se dispara un evento "submit" en él. Un controlador de JavaScript puede llamar a `preventDefault` en ese evento para deshabilitar el comportamiento predeterminado del navegador. Los elementos de campo de formulario también pueden ocurrir fuera de una etiqueta de formulario.

Cuando el usuario ha seleccionado un archivo de su sistema de archivos local en un campo de selección de archivos, la interfaz `FileReader` se puede utilizar para acceder al contenido de este archivo desde un programa JavaScript.

Los objetos `localStorage` y `sessionStorage` se pueden usar para guardar información de una manera que sobrevive a las recargas de la página. El primer objeto guarda los datos para siempre (o hasta que el usuario decida borrarlos) y el segundo los guarda hasta que se cierra el navegador.

EJERCICIOS

NEGOCIACIÓN DE CONTENIDO

Una de las cosas que HTTP puede hacer es la *negociación de contenido*. El encabezado de solicitud `Accept` se utiliza para indicar al servidor qué tipo de documento le gustaría obtener al cliente. Muchos servidores ignoran este encabezado, pero cuando un servidor conoce diversas formas de codificar un recurso, puede mirar este encabezado y enviar la que el cliente prefiera.

La URL <https://eloquentjavascript.net/author> está configurada para responder ya sea con texto sin formato, HTML o JSON, dependiendo de lo que pida el cliente. Estos formatos están identificados por los *tipos de medios* estandarizados `text/plain`, `text/html` y `application/json`.

Envía solicitudes para obtener los tres formatos de este recurso. Utiliza la propiedad `headers` en el objeto de opciones pasado a `fetch` para establecer el encabezado llamado `Accept` en el tipo de medios deseado.

Finalmente, intenta pedir el tipo de medios `application/rainbows+unicorns` y mira qué código de estado produce.

UN BANCO DE TRABAJO DE JAVASCRIPT

Construye una interfaz que permita a las personas escribir y ejecutar fragmentos de código JavaScript.

Coloca un botón al lado de un campo `<textarea>` que, al ser presionado, utilice el constructor `Function` que vimos en [Capítulo](#)

[10](#) para envolver el texto en una función y llamarla. Convierte el valor de retorno de la función, o cualquier error que genere, a una cadena y muéstralo debajo del campo de texto.

JUEGO DE LA VIDA DE CONWAY

El Juego de la vida de Conway es una simulación simple que crea “vida” artificial en una rejilla, donde cada celda puede estar viva o no. En cada generación (turno), se aplican las siguientes reglas:

- Cualquier celda viva con menos de dos o más de tres vecinos vivos muere.
- Cualquier celda viva con dos o tres vecinos vivos sigue viva en la siguiente generación.
- Cualquier celda muerta con exactamente tres vecinos vivos se convierte en una celda viva.

Un *vecino* se define como cualquier celda adyacente, incluidas las células adyacentes en diagonal.

Ten en cuenta que estas reglas se aplican a toda la rejilla de una vez, no cuadrado por cuadrado. Eso significa que el recuento de vecinos se basa en la situación al comienzo de la generación, y los cambios que ocurran en las células vecinas durante esta generación no deberían influir en el nuevo estado de una celda dada.

Implementa este juego utilizando la estructura de datos que consideres apropiada. Utiliza `Math.random` para poblar la rejilla con un patrón aleatorio inicialmente. Muestra la rejilla como una cuadrícula de campo de verificación campos, con un botón al lado

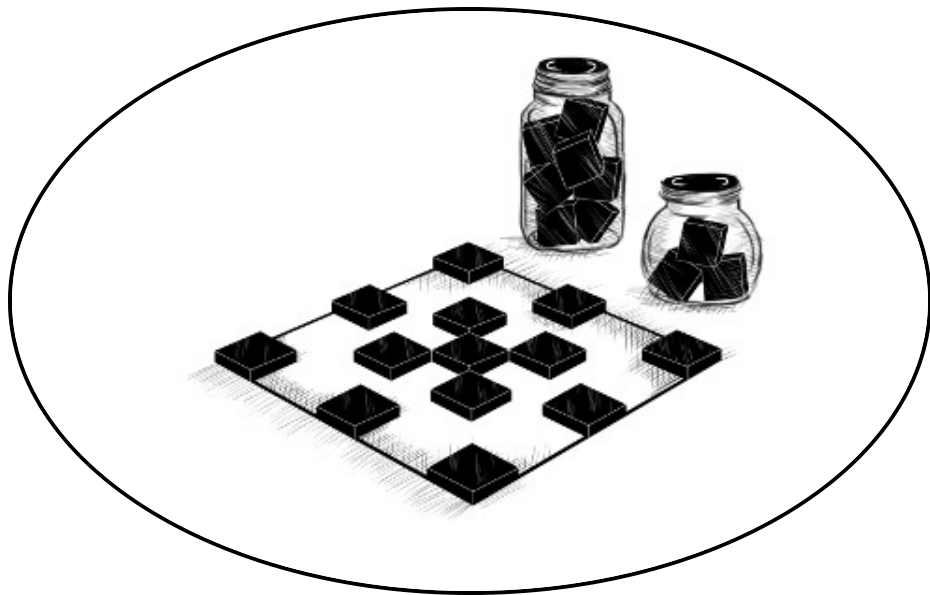
para avanzar a la siguiente generación. Cuando el usuario marque o desmarque los campos de verificación, sus cambios deberían incluirse al calcular la siguiente generación.

PROYECTO: EDITOR DE ARTE PIXELADO

```
{{meta {load_files: ["code/chapter/19_paint.js"], zip: "html include=
["css/paint.css"]}}}}
```

“Observo los muchos colores ante mí. Observo mi lienzo en blanco. Luego, intento aplicar colores como palabras que conforman poemas, como notas que conforman música.”

—Joan Miro



El material de los capítulos anteriores te brinda todos los elementos que necesitas para construir una aplicación web básica. En este capítulo, haremos precisamente eso.

Nuestra aplicación será un programa de dibujo de pixeles, donde puedes modificar una imagen píxel por píxel manipulando una vista

ampliada de la misma, mostrada como una rejilla de cuadros de colores. Puedes utilizar el programa para abrir archivos de imagen, garabatear en ellos con tu ratón u otro dispositivo señalador, y guardarlos. Así es cómo se verá:



Pintar en una computadora es genial. No necesitas preocuparte por materiales, habilidad o talento. Simplemente comienzas a manchar y ves hacia dónde llegas.

COMPONENTES

La interfaz de la aplicación muestra un gran elemento `<canvas>` en la parte superior, con varios formularios debajo de él. El usuario dibuja en la imagen seleccionando una herramienta de un campo `<select>` y luego haciendo clic, tocando o arrastrando sobre el lienzo. Hay herramientas para dibujar píxeles individuales o rectángulos, para rellenar un área y para seleccionar un color de la imagen.

Estructuraremos la interfaz del editor como un conjunto de *componentes*, objetos responsables de una parte del DOM y que pueden contener otros componentes dentro de ellos.

El estado de la aplicación consiste en la imagen actual, la herramienta seleccionada y el color seleccionado. Organizaremos las cosas de manera que el estado resida en un único valor, y los componentes de la interfaz siempre se basen en el estado actual para verse.

Para entender por qué esto es importante, consideremos la alternativa: distribuir piezas de estado a lo largo de la interfaz. Hasta cierto punto, esto es más fácil de programar. Podemos simplemente agregar un campo de color y leer su valor cuando necesitemos saber el color actual.

Pero luego agregamos el selector de colores —una herramienta que te permite hacer clic en la imagen para seleccionar el color de un píxel determinado. Para mantener el campo de color mostrando el color correcto, esa herramienta tendría que saber que el campo de color existe y actualizarlo cada vez que elige un nuevo color. Si alguna vez añades otro lugar que muestre el color (quizás el cursor del ratón podría mostrarlo), tendrías que actualizar tu código de cambio de color para mantener eso sincronizado también.

De hecho, esto crea un problema en el que cada parte de la interfaz necesita saber acerca de todas las demás partes, lo cual no es muy modular. Para aplicaciones pequeñas como la de este capítulo, eso puede no ser un problema. Para proyectos más grandes, puede convertirse en una verdadera pesadilla.

Para evitar esta pesadilla en principio, vamos a ser estrictos acerca del *flujo de datos*. Hay un estado, y la interfaz se dibuja basada en ese estado. Un componente de la interfaz puede responder a las acciones del usuario actualizando el estado, momento en el cual los

componentes tienen la oportunidad de sincronizarse con este nuevo estado.

En la práctica, cada componente se configura para que, cuando reciba un nuevo estado, también notifique a sus componentes hijos, en la medida en que estos necesiten ser actualizados. Configurar esto es un poco tedioso. Hacer que esto sea más conveniente es el principal punto de venta de muchas bibliotecas de programación para el navegador. Pero para una aplicación pequeña como esta, podemos hacerlo sin dicha infraestructura.

Las actualizaciones al estado se representan como objetos, a los que llamaremos *acciones*. Los componentes pueden crear tales acciones y *despachar* (enviarlos) a una función central de gestión de estado. Esa función calcula el próximo estado, tras lo cual los componentes de la interfaz se actualizan a este nuevo estado.

Estamos tomando la tarea desordenada de ejecutar una interfaz de usuario y aplicándole estructura. Aunque las piezas relacionadas con el DOM aún están llenas de efectos secundarios, están respaldadas por un esqueleto conceptualmente simple: el ciclo de actualización de estado. El estado determina cómo se ve el DOM, y la única forma en que los eventos del DOM pueden cambiar el estado es despachando acciones al estado.

Hay *muchas* variantes de este enfoque, cada una con sus propios beneficios y problemas, pero su idea central es la misma: los cambios de estado deben pasar por un canal único y bien definido, no suceder por todas partes.

Nuestros componentes serán clases que cumplan con una interfaz. Su constructor recibe un estado, que puede ser el estado de toda la aplicación o algún valor más pequeño si no necesita acceso a todo, y lo utiliza para construir una propiedad `dom`. Este es el elemento DOM que representa el componente. La mayoría de los constructores también tomarán otros valores que no cambiarán con el tiempo, como la función que pueden utilizar para despachar una acción.

Cada componente tiene un método `syncState` que se utiliza para sincronizarlo con un nuevo valor de estado. El método recibe un argumento, que es el estado, del mismo tipo que el primer argumento de su constructor.

EL ESTADO

El estado de la aplicación será un objeto con las propiedades `imagen`, `herramienta` y `color`. La imagen es en sí misma un objeto que almacena el ancho, alto y contenido de píxeles de la imagen. Los píxeles se almacenan en un solo array, fila por fila, de arriba abajo.

```
class Picture {
  constructor(width, height, pixels) {
    this.width = width;
    this.height = height;
    this.pixels = pixels;
  }
  static empty(width, height, color) {
    let pixels = new Array(width * height).fill(color);
    return new Picture(width, height, pixels);
  }
  pixel(x, y) {
    return this.pixels[x + y * this.width];
  }
}
```

```

draw(pixels) {
  let copy = this.pixels.slice();
  for (let {x, y, color} of pixels) {
    copy[x + y * this.width] = color;
  }
  return new Picture(this.width, this.height, copy);
}
}

```

Queremos poder tratar una imagen como un valor inmutable por razones que revisaremos más adelante en el capítulo. Pero a veces necesitamos actualizar todo un conjunto de píxeles a la vez. Para poder hacerlo, la clase tiene un método `draw` que espera un array de píxeles actualizados, objetos con propiedades `x`, `y` y `color`, y crea una nueva imagen con esos píxeles sobrescritos. Este método utiliza `slice` sin argumentos para copiar todo el array de píxeles - el inicio de la rebanada predetermina a 0, y el final predetermina a la longitud del array.

El método `empty` utiliza dos funcionalidades de array que no hemos visto antes. El constructor `Array` se puede llamar con un número para crear un array vacío de la longitud dada. El método `fill` se puede usar para llenar este array con un valor dado. Se utilizan para crear un array en el que todos los píxeles tienen el mismo color.

Los colores se almacenan como cadenas que contienen códigos de colores CSS tradicionales compuestos por un signo de almohadilla (`#`) seguido de seis dígitos hexadecimales (base-16) - dos para el componente rojo, dos para el componente verde y dos para el componente azul. Esta es una forma algo críptica e incómoda de escribir colores, pero es el formato que utiliza el campo de entrada de color HTML, y se puede usar en la propiedad `fillStyle` de un

contexto de dibujo de lienzo, por lo que para las formas en que usaremos colores en este programa, es lo bastante práctico.

El negro, donde todos los componentes son cero, se escribe como `"#000000"`, y el rosa brillante se ve como `"#ff00ff"`, donde los componentes rojo y azul tienen el valor máximo de 255, escrito `ff` en dígitos hexadecimales (que utilizan *a* a *f* para representar los dígitos 10 al 15).

Permitiremos que la interfaz envíe acciones como objetos cuyas propiedades sobrescriben las propiedades del estado anterior. El campo de color, cuando el usuario lo cambia, podría enviar un objeto como `{color: field.value}`, a partir del cual esta función de actualización puede calcular un nuevo estado.

```
function updateState(state, action) {  
  return {...state, ...action};  
}
```

Este patrón, en el que el operador de spread de objetos se utiliza primero para agregar las propiedades de un objeto existente y luego para anular algunas de ellas, es común en el código de JavaScript que utiliza objetos inmutables.

CONSTRUCCIÓN DEL DOM

Una de las principales funciones que cumplen los componentes de la interfaz es crear una estructura DOM. Nuevamente, no queremos utilizar directamente los métodos verbosos del DOM para eso, así que aquí tienes una versión ligeramente ampliada de la función `elt`:

```
function elt(type, props, ...children) {
  let dom = document.createElement(type);
  if (props) Object.assign(dom, props);
  for (let child of children) {
    if (typeof child !== "string") dom.appendChild(child);
    else dom.appendChild(document.createTextNode(child));
  }
  return dom;
}
```

La diferencia principal entre esta versión y la que usamos en [Capítulo 16](#) es que asigna *propiedades* a los nodos del DOM, no *atributos*. Esto significa que no podemos usarlo para establecer atributos arbitrarios, pero *sí* podemos usarlo para configurar propiedades cuyo valor no es una cadena, como `onclick`, que se puede establecer como una función para registrar un controlador de eventos de clic.

Esto permite este estilo conveniente para registrar manejadores de eventos:

```
<body>
  <script>
    document.body.appendChild(elt("button", {
      onclick: () => console.log("clic")
    }, "El botón"));
  </script>
</body>
```

EL LIENZO

El primer componente que definiremos es la parte de la interfaz que muestra la imagen como una cuadrícula de cuadros coloreados. Este componente es responsable de dos cosas: mostrar una imagen y

comunicar evento de punteros en esa imagen al resto de la aplicación.

Como tal, podemos definirlo como un componente que solo conoce la imagen actual, no todo el estado de la aplicación. Dado que no sabe cómo funciona la aplicación en su totalidad, no puede despachar acciones directamente. Más bien, al responder a eventos de puntero, llama a una función de devolución de llamada proporcionada por el código que lo creó, que se encargará de las partes específicas de la aplicación.

```
const scale = 10;

class PictureCanvas {
  constructor(image, pointerDown) {
    this.dom = elt("canvas", {
      onmousedown: event => this.mouse(event, pointerDown),
      ontouchstart: event => this.touch(event, pointerDown)
    });
    this.syncState(image);
  }
  syncState(image) {
    if (this.image === image) return;
    this.image = image;
    drawPicture(this.image, this.dom, scale);
  }
}
```

Dibujamos cada píxel como un cuadrado de 10 por 10, según lo determinado por la constante `scale`. Para evitar trabajo innecesario, el componente realiza un seguimiento de su imagen actual y solo vuelve a dibujar cuando se le proporciona una nueva imagen a `syncState`.

La función de dibujo real establece el tamaño del lienzo en función de la escala y el tamaño de la imagen y lo llena con una serie de cuadrados, uno para cada píxel.

```
function drawPicture(picture, canvas, scale) {
  canvas.width = picture.width * scale;
  canvas.height = picture.height * scale;
  let cx = canvas.getContext("2d");

  for (let y = 0; y < picture.height; y++) {
    for (let x = 0; x < picture.width; x++) {
      cx.fillStyle = picture.pixel(x, y);
      cx.fillRect(x * scale, y * scale, scale, scale);
    }
  }
}
```

Cuando se presiona el botón izquierdo del mouse mientras está sobre el lienzo de la imagen, el componente llama al callback `pointerDown`, dándole la posición del píxel que se hizo clic, en coordenadas de la imagen. Esto se usará para implementar la interacción del mouse con la imagen. El callback puede devolver otra función de callback para ser notificado cuando se mueve el puntero a un píxel diferente mientras se mantiene presionado el botón.

```
PictureCanvas.prototype.mouse = function(downEvent, onDown) {
  if (downEvent.button !== 0) return;
  let pos = pointerPosition(downEvent, this.dom);
  let onMove = onDown(pos);
  if (!onMove) return;
  let move = moveEvent => {
    if (moveEvent.buttons === 0) {
      this.dom.removeEventListener("mousemove", move);
    } else {
      let newPos = pointerPosition(moveEvent, this.dom);
      if (newPos.x === pos.x && newPos.y === pos.y) return;
      pos = newPos;
    }
  };
  this.dom.addEventListener("mousemove", move);
}
```

```

        onMove(newPos);
    }
};
this.dom.addEventListener("mousemove", move);
};

function pointerPosition(pos, domNode) {
    let rect = domNode.getBoundingClientRect();
    return {x: Math.floor((pos.clientX - rect.left) / scale),
            y: Math.floor((pos.clientY - rect.top) / scale)};
}

```

Dado que conocemos el tamaño de los píxeles y podemos usar `getBoundingClientRect` para encontrar la posición del lienzo en la pantalla, es posible ir desde las coordenadas del evento del mouse (`clientX` y `clientY`) hasta las coordenadas de la imagen. Estas siempre se redondean hacia abajo para que se refieran a un píxel específico.

Con eventos táctiles, tenemos que hacer algo similar, pero utilizando diferentes eventos y asegurándonos de llamar a `preventDefault` en el evento `"touchstart"` para evitar el desplazamiento.

```

PictureCanvas.prototype.touch = function(startEvent,
                                          onDown) {
    let pos = pointerPosition(startEvent.touches[0], this.dom);
    let onMove = onDown(pos);
    startEvent.preventDefault();
    if (!onMove) return;
    let move = moveEvent => {
        let newPos = pointerPosition(moveEvent.touches[0],
                                     this.dom);
        if (newPos.x == pos.x && newPos.y == pos.y) return;
        pos = newPos;
        onMove(newPos);
    };
    let end = () => {
        this.dom.removeEventListener("touchmove", move);
    };
}

```

```
        this.dom.removeEventListener("touchend", end);  
    };  
    this.dom.addEventListener("touchmove", move);  
    this.dom.addEventListener("touchend", end);  
};
```

Para eventos táctiles, `clientX` y `clientY` no están disponibles directamente en el objeto de evento, pero podemos usar las coordenadas del primer objeto táctil en la propiedad `touches`.

LA APLICACIÓN

Para hacer posible construir la aplicación pieza por pieza, implementaremos el componente principal como una cáscara alrededor de un lienzo de imagen y un conjunto dinámico de tools y controls que pasamos a su constructor.

Los *controles* son los elementos de interfaz que aparecen debajo de la imagen. Se proporcionarán como un array de constructores de component.

Las *herramientas* hacen cosas como dibujar píxeles o rellenar un área. La aplicación muestra el conjunto de herramientas disponibles como un campo `<select>`. La herramienta actualmente seleccionada determina qué sucede cuando el usuario interactúa con la imagen con un dispositivo puntero. El conjunto de herramientas disponibles se proporciona como un objeto que mapea los nombres que aparecen en el campo desplegable a funciones que implementan las herramientas. Dichas funciones reciben como argumentos una posición de imagen, un estado de aplicación actual y una función `dispatch`. Pueden devolver una función manejadora de

movimiento que se llama con una nueva posición y un estado actual cuando el puntero se mueve a un píxel diferente.

```
class PixelEditor {
  constructor(state, config) {
    let {tools, controls, dispatch} = config;
    this.state = state;

    this.canvas = new PictureCanvas(state.picture, pos => {
      let tool = tools[this.state.tool];
      let onMove = tool(pos, this.state, dispatch);
      if (onMove) return pos => onMove(pos, this.state);
    });
    this.controls = controls.map(
      Control => new Control(state, config));
    this.dom = elt("div", {}, this.canvas.dom, elt("br"),
      ...this.controls.reduce(
        (a, c) => a.concat(" ", c.dom), []));
  }
  syncState(state) {
    this.state = state;
    this.canvas.syncState(state.picture);
    for (let ctrl of this.controls) ctrl.syncState(state);
  }
}
```

El manejador de puntero dado a `PictureCanvas` llama a la herramienta actualmente seleccionada con los argumentos apropiados y, si eso devuelve un manejador de movimiento, lo adapta para también recibir el estado.

Todos los controles se construyen y almacenan en `this.controls` para que puedan actualizarse cuando cambie el estado de la aplicación. La llamada a `reduce` introduce espacios entre los elementos DOM de los controles. De esa manera, no se ven tan juntos.

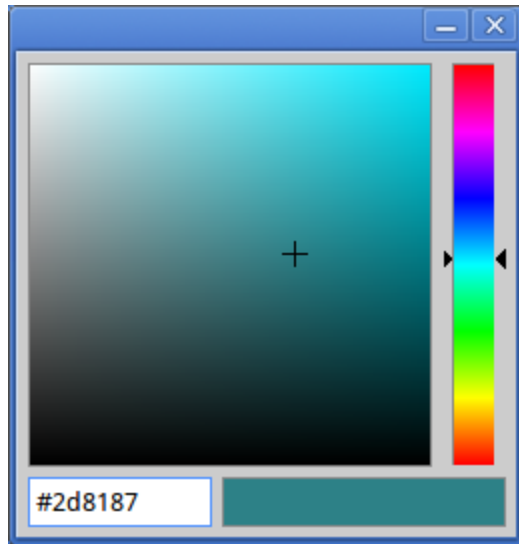
El primer control es el menú de selección de tool. Crea un elemento `<select>` con una opción para cada herramienta y configura un manejador de evento "change" que actualiza el estado de la aplicación cuando el usuario selecciona una herramienta diferente.

```
class ToolSelect {
  constructor(state, {tools, dispatch}) {
    this.select = elt("select", {
      onchange: () => dispatch({tool: this.select.value})
    }, ...Object.keys(tools).map(name => elt("option", {
      selected: name == state.tool
    }, name)));
    this.dom = elt("label", null, "✂ Herramienta: ",
this.select);
  }
  syncState(state) { this.select.value = state.tool; }
}
```

Al envolver el texto de la etiqueta y el campo en un elemento `<label>`, le decimos al navegador que la etiqueta pertenece a ese campo para que, por ejemplo, se pueda hacer clic en la etiqueta para enfocar el campo.

También necesitamos poder cambiar el color, así que agreguemos un control para eso. Un elemento HTML `<input>` con un atributo `type` de `color` nos brinda un campo de formulario especializado para seleccionar colores. El valor de dicho campo siempre es un código de color CSS en formato `"#RRGGBB"` (componentes rojo, verde y azul, dos dígitos por color). El navegador mostrará una interfaz de selector de color cuando el usuario interactúe con él.

Dependiendo del navegador, el selector de color puede lucir así:



Este control crea un campo de ese tipo y lo conecta para que se mantenga sincronizado con la propiedad `color` del estado de la aplicación.

```
class ColorSelect {
  constructor(state, {dispatch}) {
    this.input = elt("input", {
      type: "color",
      value: state.color,
      onchange: () => dispatch({color: this.input.value})
    });
    this.dom = elt("label", null, "🎨 Color: ", this.input);
  }
  syncState(state) { this.input.value = state.color; }
}
```

HERRAMIENTAS DE DIBUJO

Antes de poder dibujar algo, necesitamos implementar las herramientas que controlarán la funcionalidad de eventos de ratón o táctiles en el lienzo.

La herramienta más básica es la herramienta de dibujo, que cambia cualquier píxel en el que hagas clic o toques al color seleccionado actualmente. Envía una acción que actualiza la imagen a una versión en la que el píxel señalado recibe el color seleccionado actualmente.

```
function draw(pos, state, dispatch) {  
  function drawPixel({x, y}, state) {  
    let drawn = {x, y, color: state.color};  
    dispatch({picture: state.picture.draw([drawn])});  
  }  
  drawPixel(pos, state);  
  return drawPixel;  
}
```

La función llama inmediatamente a la función `drawPixel`, pero también la devuelve para que sea llamada nuevamente para los píxeles recién tocados cuando el usuario arrastra o desliza sobre la imagen.

Para dibujar formas más grandes, puede ser útil crear rápidamente rectángulos. La herramienta `rectángulo` dibuja un rectángulo entre el punto donde comienzas a arrastrar y el punto al que arrastras.

```
function rectangle(start, state, dispatch) {  
  function drawRectangle(pos) {  
    let xStart = Math.min(start.x, pos.x);  
    let yStart = Math.min(start.y, pos.y);  
    let xEnd = Math.max(start.x, pos.x);  
    let yEnd = Math.max(start.y, pos.y);  
    let drawn = [];  
    for (let y = yStart; y <= yEnd; y++) {  
      for (let x = xStart; x <= xEnd; x++) {  
        drawn.push({x, y, color: state.color});  
      }  
    }  
  }  
}
```

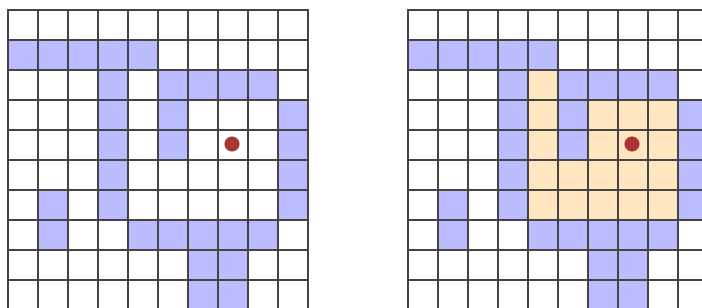
```

    dispatch({picture: state.picture.draw(drawn)});
  }
  drawRectangle(start);
  return drawRectangle;
}

```

Un detalle importante en esta implementación es que al arrastrar, el rectángulo se vuelve a dibujar en la imagen a partir del estado *original*. De esta manera, puedes hacer que el rectángulo sea más grande o más pequeño nuevamente mientras lo creas, sin que los rectángulos intermedios queden pegados en la imagen final. Esta es una de las razones por las que los objetos de imagen inmutables son útiles; veremos otra razón más adelante.

Implementar el relleno por inundación es algo más complejo. Se trata de una herramienta que llena el píxel bajo el puntero y todos los píxeles adyacentes que tengan el mismo color. “Adyacente” significa adyacente directamente en horizontal o vertical, no diagonalmente. Esta imagen ilustra el conjunto de píxeles coloreados cuando se utiliza la herramienta de relleno por inundación en el píxel marcado:



Curiosamente, la forma en que lo haremos se parece un poco al código de búsqueda de caminos del [Capítulo 7](#). Mientras que ese código buscaba a través de un grafo para encontrar una ruta, este código busca a través de una cuadrícula para encontrar todos los

píxeles “conectados”. El problema de llevar un conjunto ramificado de rutas posibles es similar.

```
const alrededor = [{dx: -1, dy: 0}, {dx: 1, dy: 0},
                  {dx: 0, dy: -1}, {dx: 0, dy: 1}];

function rellenar({x, y}, estado, despachar) {
  let colorObjetivo = estado.imagen.pixel(x, y);
  let dibujados = [{x, y, color: estado.color}];
  let visitados = new Set();
  for (let hecho = 0; hecho < dibujados.length; hecho++) {
    for (let {dx, dy} of alrededor) {
      let x = dibujados[hecho].x + dx, y = dibujados[hecho].y +
dy;
      if (x >= 0 && x < estado.imagen.ancho &&
          y >= 0 && y < estado.imagen.alto &&
          !visitados.has(x + "," + y) &&
          estado.imagen.pixel(x, y) == colorObjetivo) {
        dibujados.push({x, y, color: estado.color});
        visitados.add(x + "," + y);
      }
    }
  }
  despachar({imagen: estado.imagen.dibujar(dibujados)});
}
```

El array de píxeles dibujados funciona como la lista de trabajo de la función. Para cada píxel alcanzado, tenemos que ver si algún píxel adyacente tiene el mismo color y aún no ha sido pintado. El contador del bucle va rezagado respecto a la longitud del array dibujados a medida que se añaden nuevos píxeles. Cualquier píxel por delante de él aún necesita ser explorado. Cuando alcanza la longitud, no quedan píxeles sin explorar y la función termina.

La última herramienta es un selector de color, que te permite apuntar a un color en la imagen para usarlo como color de dibujo actual.

```
function seleccionar(pos, estado, despachar) {
  despachar({color: estado.imagen.pixel(pos.x, pos.y)});
}
```## Guardar y cargar
```

Cuando hemos dibujado nuestra obra maestra, queríamos guardarla para más tarde. Deberíamos añadir un botón para descargar la imagen actual como un archivo de imagen. Este control proporciona ese botón:

```
```{includeCode: true}
class SaveButton {
  constructor(state) {
    this.picture = state.picture;
    this.dom = elt("button", {
      onclick: () => this.save()
    }, "🖼 Guardar");
  }
  save() {
    let canvas = elt("canvas");
    drawPicture(this.picture, canvas, 1);
    let link = elt("a", {
      href: canvas.toDataURL(),
      download: "pixelart.png"
    });
    document.body.appendChild(link);
    link.click();
    link.remove();
  }
  syncState(state) { this.picture = state.picture; }
}
```

El componente lleva un registro de la imagen actual para que pueda acceder a ella al guardar. Para crear el archivo de imagen, utiliza un elemento `<canvas>` en el que dibuja la imagen (a una escala de un píxel por píxel).

El método `toDataURL` en un elemento `canvas` crea una URL que empieza con `data:`. A diferencia de las URL `http:` y `https:`, las

URL de datos contienen todo el recurso en la URL. Por lo general, son muy largas, pero nos permiten crear enlaces funcionales a imágenes arbitrarias aquí mismo en el navegador.

Para realmente hacer que el navegador descargue la imagen, luego creamos un elemento de enlace que apunta a esta URL y tiene un atributo `download`. Tales enlaces, al hacer clic en ellos, muestran un cuadro de diálogo para guardar el archivo en el navegador. Añadimos ese enlace al documento, simulamos un clic en él y luego lo eliminamos. Se pueden hacer muchas cosas con la tecnología del navegador, pero a veces la forma de hacerlo es bastante extraña.

Y la cosa se pone peor. También queríamos cargar archivos de imagen existentes en nuestra aplicación. Para hacer eso, nuevamente definimos un componente de botón.

```
class LoadButton {
  constructor(_, {dispatch}) {
    this.dom = elt("button", {
      onclick: () => startLoad(dispatch)
    }, "📁 Cargar");
  }
  syncState() {}
}

function startLoad(dispatch) {
  let input = elt("input", {
    type: "file",
    onchange: () => finishLoad(input.files[0], dispatch)
  });
  document.body.appendChild(input);
  input.click();
  input.remove();
}
```


Para acceder a un archivo en la computadora del usuario, necesitamos que el usuario seleccione el archivo a través de un campo de entrada de archivo. Pero no quiero que el botón de carga se vea como un campo de entrada de archivo, así que creamos el campo de entrada de archivo cuando se hace clic en el botón y luego fingimos que este campo de entrada de archivo fue clicado.

Cuando el usuario ha seleccionado un archivo, podemos usar `FileReader` para acceder a su contenido, nuevamente como una URL de datos. Esa URL se puede utilizar para crear un elemento ``, pero debido a que no podemos acceder directamente a los píxeles en una imagen de ese tipo, no podemos crear un objeto `Picture` a partir de eso.

```
function finishLoad(file, dispatch) {
  if (file == null) return;
  let reader = new FileReader();
  reader.addEventListener("load", () => {
    let image = elt("img", {
      onload: () => dispatch({
        picture: pictureFromImage(image)
      }),
      src: reader.result
    });
  });
  reader.readAsDataURL(file);
}
```

Para acceder a los píxeles, primero debemos dibujar la imagen en un elemento `<canvas>`. El contexto del canvas tiene un método `getImageData` que permite a un script leer sus píxeles. Por lo tanto, una vez que la imagen esté en el canvas, podemos acceder a ella y construir un objeto `Picture`.

```
function pictureFromImage(image) {
  let width = Math.min(100, image.width);
  let height = Math.min(100, image.height);
  let canvas = elt("canvas", {width, height});
  let cx = canvas.getContext("2d");
  cx.drawImage(image, 0, 0);
  let pixels = [];
  let {data} = cx.getImageData(0, 0, width, height);

  function hex(n) {
    return n.toString(16).padStart(2, "0");
  }
  for (let i = 0; i < data.length; i += 4) {
    let [r, g, b] = data.slice(i, i + 3);
    pixels.push("#" + hex(r) + hex(g) + hex(b));
  }
  return new Picture(width, height, pixels);
}
```

Limitaremos el tamaño de las imágenes a 100 por 100 píxeles, ya que cualquier cosa más grande se verá *enorme* en nuestra pantalla y podría ralentizar la interfaz.

La propiedad `data` del objeto devuelto por `getImageData` es un array de componentes de color. Para cada píxel en el rectángulo especificado por los argumentos, contiene cuatro valores, que representan los componentes rojo, verde, azul y *alfa* del color del píxel, como números entre 0 y 255. La parte alfa representa la opacidad: cuando es cero, el píxel es totalmente transparente, y cuando es 255, es totalmente opaco. Para nuestro propósito, podemos ignorarla.

Los dos dígitos hexadecimales por componente, como se usa en nuestra notación de color, corresponden precisamente al rango del 0 al 255: dos dígitos en base 16 pueden expresar $16^2 = 256$ números

diferentes. El método `toString` de los números puede recibir como argumento una base, por lo que `n.toString(16)` producirá una representación en cadena en base 16. Debemos asegurarnos de que cada número tenga dos dígitos, por lo que la función auxiliar `hex` llama a `padStart` para agregar un cero inicial cuando sea necesario.

¡Ya podemos cargar y guardar! Eso deja una característica más antes de que hayamos terminado.

HISTORIAL DE DESHACER

La mitad del proceso de edición consiste en cometer pequeños errores y corregirlos. Por lo tanto, una característica importante en un programa de dibujo es un historial de deshacer.

Para poder deshacer cambios, necesitamos almacenar versiones anteriores de la imagen. Dado que es un valor inmutable, eso es fácil. Pero sí requiere un campo adicional en el estado de la aplicación.

Agregaremos una matriz `done` para mantener versiones anteriores de la imagen. Mantener esta propiedad requiere una función de actualización de estado más complicada que añade imágenes a la matriz.

Pero no queremos almacenar *cada* cambio, solo los cambios que ocurran en un determinado espacio de tiempo. Para poder hacer eso, necesitaremos una segunda propiedad, `doneAt`, que rastree la hora en la que almacenamos por última vez una imagen en el historial.

```
function historyUpdateState(state, action) {  
  if (action.undo == true) {
```

```

    if (state.done.length == 0) return state;
    return {
        ...state,
        picture: state.done[0],
        done: state.done.slice(1),
        doneAt: 0
    };
} else if (action.picture &&
           state.doneAt < Date.now() - 1000) {
    return {
        ...state,
        ...action,
        done: [state.picture, ...state.done],
        doneAt: Date.now()
    };
} else {
    return {...state, ...action};
}
}

```

Cuando la acción es una acción de deshacer, la función toma la imagen más reciente del historial y la convierte en la imagen actual. Establece `doneAt` en cero para garantizar que el siguiente cambio almacenará la imagen nuevamente en el historial, permitiéndote revertir a ella en otro momento si lo deseas.

De lo contrario, si la acción contiene una nueva imagen y la última vez que almacenamos algo fue hace más de un segundo (1000 milisegundos), las propiedades `done` y `doneAt` se actualizan para almacenar la imagen anterior.

El botón de deshacer componente no hace mucho. Despacha acciones de deshacer al hacer clic y se deshabilita cuando no hay nada que deshacer.

```

class UndoButton {
    constructor(state, {dispatch}) {

```

```

    this.dom = elt("button", {
      onclick: () => dispatch({undo: true}),
      disabled: state.done.length == 0
    }, "↶ Deshacer");
  }
  syncState(state) {
    this.dom.disabled = state.done.length == 0;
  }
}

```

VAMOS A DIBUJAR

Para configurar la aplicación, necesitamos crear un estado, un conjunto de herramientas, un conjunto de controles y una función despachar. Podemos pasarlos al constructor `PixelEditor` para crear el componente principal. Dado que necesitaremos crear varios editores en los ejercicios, primero definimos algunos enlaces.

```

const startState = {
  tool: "draw",
  color: "#000000",
  picture: Picture.empty(60, 30, "#f0f0f0"),
  done: [],
  doneAt: 0
};

const baseTools = {draw, fill, rectangle, pick};

const baseControls = [
  ToolSelect, ColorSelect, SaveButton, LoadButton, UndoButton
];

function startPixelEditor({state = startState,
                          tools = baseTools,
                          controls = baseControls}) {
  let app = new PixelEditor(state, {
    tools,
    controls,
    dispatch(action) {

```

```

        state = historyUpdateState(state, action);
        app.syncState(state);
    }
});
return app.dom;
}

```

Cuando desestructuras un objeto o un array, puedes usar `=` después de un nombre de enlace para darle al enlace un valor predeterminado, que se usa cuando la propiedad está ausente o tiene `undefined`. La función `startPixelEditor` hace uso de esto para aceptar un objeto con varias propiedades opcionales como argumento. Si, por ejemplo, no proporcionas una propiedad `tools`, entonces `tools` estará vinculado a `baseTools`. Así es como obtenemos un editor real en la pantalla:

```

<div></div>
<script>
    document.querySelector("div")
        .appendChild(startPixelEditor({}));
</script>

```

¿POR QUÉ ES TAN DIFÍCIL?

La tecnología del navegador es asombrosa. Proporciona un poderoso conjunto de bloques de construcción de interfaz, formas de diseñar y manipularlos, y herramientas para inspeccionar y depurar tus aplicaciones. El software que escribes para el navegador puede ejecutarse en casi todas las computadoras y teléfonos del planeta.

Al mismo tiempo, la tecnología del navegador es ridícula. Tienes que aprender una gran cantidad de trucos tontos y hechos oscuros para dominarla, y el modelo de programación predeterminado que ofrece es tan problemático que la mayoría de los programadores prefieren

cubrirlo con varias capas de abstracción en lugar de lidiar con él directamente.

Y aunque la situación definitivamente está mejorando, en su mayoría lo hace en forma de más elementos que se agregan para abordar deficiencias, creando aún más complejidad. Una característica utilizada por un millón de sitios web realmente no se puede reemplazar. Incluso si se pudiera, sería difícil decidir con qué debiera ser reemplazada.

La tecnología nunca existe en un vacío; estamos limitados por nuestras herramientas y los factores sociales, económicos e históricos que las produjeron. Esto puede ser molesto, pero generalmente es más productivo tratar de construir una buena comprensión de cómo funciona la realidad técnica *existente* y por qué es como es, que luchar contra ella o esperar otra realidad.

Nuevas abstracciones *pueden* ser útiles. El modelo de componente y la convención de flujo de datos que utilicé en este capítulo es una forma rudimentaria de eso. Como se mencionó, hay bibliotecas que intentan hacer la programación de interfaces de usuario más agradable. En el momento de escribir esto, [React](#) y [Svelte](#) son opciones populares, pero hay toda una industria de tales marcos. Si estás interesado en programar aplicaciones web, recomiendo investigar algunos de ellos para comprender cómo funcionan y qué beneficios proporcionan.

EJERCICIOS

Todavía hay espacio para mejorar nuestro programa. Vamos a agregar algunas funciones más como ejercicios.

ATAJOS DE TECLADO

Agrega atajos de teclado a la aplicación. La primera letra del nombre de una herramienta selecciona la herramienta, y `CONTROL-Z` o `COMMAND-Z` activa el deshacer.

Haz esto modificando el componente `PixelEditor`. Agrega una propiedad `tabIndex` de 0 al elemento `<div>` envolvente para que pueda recibir el enfoque del teclado. Ten en cuenta que la *propiedad* correspondiente al atributo `tabindex` se llama `tabIndex`, con una `I` mayúscula, y nuestra función `elt` espera nombres de propiedades. Registra los manejadores de eventos de teclas directamente en ese elemento. Esto significa que debes hacer clic, tocar o moverte al tabulador en la aplicación antes de poder interactuar con el teclado.

Recuerda que los eventos de teclado tienen las propiedades `ctrlKey` y `metaKey` (para la tecla `COMMAND` en Mac) que puedes utilizar para ver si esas teclas están presionadas.

DIBUJANDO EFICIENTEMENTE

Durante el dibujo, la mayoría del trabajo que hace nuestra aplicación ocurre en `drawPicture`. Crear un nuevo estado y actualizar el resto del DOM no es muy costoso, pero repintar todos los píxeles en el lienzo es bastante trabajo.

Encuentra una forma de hacer que el método `syncState` de `PictureCanvas` sea más rápido redibujando solo los píxeles que realmente cambiaron.

Recuerda que `drawPicture` también es utilizado por el botón de guardar, así que si lo cambias, asegúrate de que los cambios no rompan el uso anterior o crea una nueva versión con un nombre diferente.

También ten en cuenta que al cambiar el tamaño de un elemento `<canvas>`, establecer sus propiedades `width` o `height`, lo borra y lo vuelve completamente transparente nuevamente.

CÍRCULOS

Define una herramienta llamada `circle` que dibuje un círculo relleno cuando arrastres. El centro del círculo se encuentra en el punto donde comienza el gesto de arrastre o toque, y su radio está determinado por la distancia arrastrada.

LÍNEAS ADECUADAS

Este es un ejercicio más avanzado que los dos anteriores, y requerirá que diseñes una solución a un problema no trivial. Asegúrate de tener mucho tiempo y paciencia antes de comenzar a trabajar en este ejercicio, y no te desanimes por los fallos iniciales.

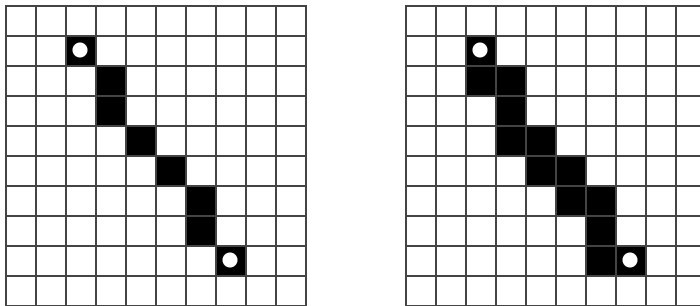
En la mayoría de los navegadores, al seleccionar la herramienta de dibujo y arrastrar rápidamente sobre la imagen, no obtienes una línea cerrada. En su lugar, obtienes puntos con huecos entre ellos porque los eventos `"mousemove"` o `"touchmove"` no se dispararon lo suficientemente rápido como para alcanzar cada píxel.

Mejora la herramienta de dibujo para que dibuje una línea completa. Esto significa que debes hacer que la función de

controlador de movimiento recuerde la posición anterior y la conecte con la actual.

Para hacer esto, dado que los píxeles pueden estar a una distancia arbitraria, tendrás que escribir una función general de dibujo de líneas.

Una línea entre dos píxeles es una cadena conectada de píxeles, lo más recta posible, que va desde el comienzo hasta el final. Los píxeles diagonalmente adyacentes cuentan como conectados. Por lo tanto, una línea inclinada debería verse como la imagen de la izquierda, no como la de la derecha.

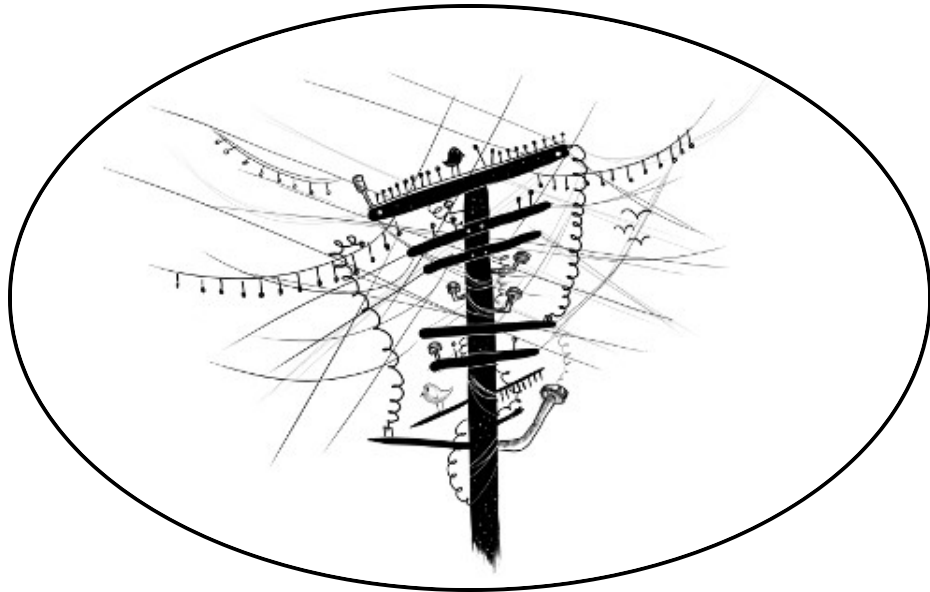


Finalmente, si tenemos código que dibuja una línea entre dos puntos arbitrarios, podríamos usarlo también para definir una herramienta de línea, que dibuja una línea recta entre el inicio y el final de un arrastre.

NODE.JS

“Un estudiante preguntó: “Los programadores de antaño solo usaban máquinas simples y ningún lenguaje de programación, sin embargo, creaban programas hermosos. ¿Por qué nosotros usamos máquinas complicadas y lenguajes de programación?”. Fu-Tzu respondió: “Los constructores de antaño solo usaban palos y arcilla, sin embargo, creaban hermosas chozas.””

—Maestro Yuan-Ma, *El Libro de la Programación*



Hasta ahora, hemos utilizado el lenguaje JavaScript en un solo entorno: el navegador. Este capítulo y el [siguiente](#) introducirán brevemente Node.js, un programa que te permite aplicar tus habilidades con JavaScript fuera del navegador. Con él, puedes construir desde pequeñas herramientas de línea de comandos hasta servidores HTTP server que alimentan sitios web dinámicos.

Estos capítulos tienen como objetivo enseñarte los conceptos principales que Node.js utiliza y darte información suficiente para escribir programas útiles para él. No intentan ser un tratamiento completo, ni siquiera exhaustivo, de la plataforma.

Si deseas seguir y ejecutar el código en este capítulo, necesitarás instalar Node.js versión 18 o superior. Para hacerlo, ve a <https://nodejs.org> y sigue las instrucciones de instalación para tu sistema operativo. También puedes encontrar más documentación para Node.js allí.

ANTECEDENTES

Cuando se construyen sistemas que se comunican a través de la red, la forma en que gestionas la entrada y el output—es decir, la lectura y escritura de datos desde y hacia la red y el disco duro—puede marcar una gran diferencia en cuán rápido responde un sistema al usuario o a las solicitudes de red.

En tales programas, la programación asincrónica a menudo es útil. Permite que el programa envíe y reciba datos desde y hacia múltiples dispositivos al mismo tiempo sin una complicada gestión de hilos y sincronización.

Node fue concebido inicialmente con el propósito de hacer que la programación asincrónica sea fácil y conveniente. JavaScript se presta bien a un sistema como Node. Es uno de los pocos lenguajes de programación que no tiene una forma incorporada de manejar la entrada y salida. Por lo tanto, JavaScript podría adaptarse al enfoque algo excéntrico de Node para la programación de red y sistemas de archivos sin terminar con dos interfaces inconsistentes. En 2009,

cuando se diseñaba Node, la gente ya estaba realizando programación basada en callbacks en el navegador, por lo que la comunidad alrededor del lenguaje estaba acostumbrada a un estilo de programación asíncrona.

EL COMANDO NODE

Cuando Node.js está instalado en un sistema, proporciona un programa llamado `node`, que se utiliza para ejecutar archivos de JavaScript. Supongamos que tienes un archivo `hello.js`, que contiene este código:

```
let message = "Hola mundo";  
console.log(message);
```

Luego puedes ejecutar `node` desde la línea de comandos de la siguiente manera para ejecutar el programa:

```
$ node hello.js  
Hola mundo
```

El método `console.log` en Node hace algo similar a lo que hace en el navegador. Imprime un texto. Pero en Node, el texto irá al flujo de salida estándar del proceso, en lugar de ir a la consola de JavaScript de un navegador. Al ejecutar `node` desde la línea de comandos, significa que verás los valores registrados en tu terminal.

Si ejecutas `node` sin proporcionarle un archivo, te proporcionará un indicador en el que puedes escribir código JavaScript y ver inmediatamente el resultado.

```
$ node  
> 1 + 1
```

```
2
> [-1, -2, -3].map(Math.abs)
[1, 2, 3]
> process.exit(0)
$
```

El enlace `process`, al igual que el enlace `console`, está disponible globalmente en Node. Proporciona varias formas de inspeccionar y manipular el programa actual. El método `exit` finaliza el proceso y puede recibir un código de estado de salida, que le indica al programa que inició node (en este caso, la shell de línea de comandos) si el programa se completó correctamente (código cero) o si se encontró un error (cualquier otro código).

Para encontrar los argumentos de línea de comandos dados a tu script, puedes leer `process.argv`, que es un array de cadenas. Ten en cuenta que también incluye el nombre del comando `node` y el nombre de tu script, por lo que los argumentos reales comienzan en el índice 2. Si `showargv.js` contiene la instrucción `console.log(process.argv)`, podrías ejecutarlo de la siguiente manera:

```
$ node showargv.js one --and two
["node", "/tmp/showargv.js", "one", "--and", "two"]
```

Todos los enlaces globales de JavaScript estándar, como `Array`, `Math` y `JSON`, también están presentes en el entorno de Node. La funcionalidad relacionada con el navegador, como `document` o `prompt`, no lo está.

MÓDULOS

Además de los enlaces que mencioné, como `console` y `process`, Node agrega pocos enlaces adicionales en el ámbito global. Si deseas

acceder a funcionalidades integradas, debes solicitarlas al sistema de módulos.

Node comenzó utilizando el sistema de módulos CommonJS, basado en la función `require`, que vimos en [Capítulo 10](#). Aún utilizará este sistema de forma predeterminada cuando cargues un archivo `.js`.

Pero también soporta el sistema de módulos ES más moderno. Cuando el nombre de un script termina en `.mjs`, se considera que es un módulo de este tipo, y puedes usar `import` y `export` en él (pero no `require`). Utilizaremos módulos ES en este capítulo.

Cuando se importa un módulo, ya sea con `require` o `import`, Node debe resolver la cadena proporcionada a un archivo real que pueda cargar. Los nombres que comienzan con `/`, `./` o `../` se resuelven como archivos, relativos a la ruta del módulo actual. Aquí, `.` representa el directorio actual, `../` para un directorio arriba, y `/` para la raíz del sistema de archivos. Por lo tanto, si solicitas `"/graph.mjs"` desde el archivo `/tmp/robot/robot.mjs`, Node intentará cargar el archivo `/tmp/robot/graph.mjs`.

Cuando se importa una cadena que no parece una ruta relativa o absoluta, se asume que se refiere a un módulo integrado o un módulo instalado en un directorio `node_modules`. Por ejemplo, importar desde `"node:fs"` te dará el módulo integrado del sistema de archivos de Node. E importar `"robot"` podría intentar cargar la biblioteca encontrada en `node_modules/robot/`. Una forma común de instalar estas bibliotecas es usando NPM, a lo cual volveremos en un momento.

Configuremos un proyecto pequeño que consta de dos archivos. El primero, llamado `main.mjs`, define un script que puede ser llamado desde la línea de comandos para revertir una cadena.

```
import {reverse} from "./reverse.mjs";

// El índice 2 contiene el primer argumento real de la línea de
comandos
let argument = process.argv[2];

console.log(reverse(argument));
```

El archivo `reverse.mjs` define una biblioteca para revertir cadenas, que puede ser utilizada tanto por esta herramienta de línea de comandos como por otros scripts que necesiten acceso directo a una función para revertir cadenas.

```
export function reverse(string) {
  return Array.from(string).reverse().join("");
}
```

Recuerda que `export` se utiliza para declarar que un enlace es parte de la interfaz del módulo. Eso permite que `main.mjs` importe y utilice la función.

Ahora podemos llamar a nuestra herramienta de esta manera:

```
$ node main.mjs JavaScript
tpircSavaJ
```

INSTALANDO CON NPM

NPM, que fue introducido en [Capítulo 10](#), es un repositorio en línea de módulos de JavaScript, muchos de los cuales están escritos

específicamente para Node. Cuando instalas Node en tu computadora, también obtienes el comando `npm`, que puedes usar para interactuar con este repositorio.

El uso principal de NPM es descargar paquetes. Vimos el paquete `ini` en [Capítulo 10](#). Podemos usar NPM para buscar e instalar ese paquete en nuestra computadora.

```
$ npm install ini
agregado 1 paquete en 723ms

$ node
> const {parse} = require("ini");
> parse("x = 1\ny = 2");
{ x: '1', y: '2' }
```

Después de ejecutar `npm install`, NPM habrá creado un directorio llamado `node_modules`. Dentro de ese directorio estará un directorio `ini` que contiene la biblioteca. Puedes abrirlo y ver el código. Cuando importamos `"ini"`, esta biblioteca se carga, y podemos llamar a su propiedad `parse` para analizar un archivo de configuración. Por defecto, NPM instala paquetes en el directorio actual, en lugar de en un lugar centralizado. Si estás acostumbrado a otros gestores de paquetes, esto puede parecer inusual, pero tiene ventajas: pone a cada aplicación en control total de los paquetes que instala y facilita la gestión de versiones y limpieza al eliminar una aplicación.

ARCHIVOS DE PAQUETE

Después de ejecutar `npm install` para instalar algún paquete, encontrarás no solo un directorio `node_modules`, sino también un archivo llamado `package.json` en tu directorio actual. Se

recomienda tener tal archivo para cada proyecto. Puedes crearlo manualmente o ejecutar `npm init`. Este archivo contiene información sobre el proyecto, como su nombre y versión, y enumera sus dependencias.

La simulación del robot de [Capítulo 7](#), modularizada en el ejercicio en [Capítulo 10](#), podría tener un archivo `package.json` como este:

```
{
  "author": "Marijn Haverbeke",
  "name": "eloquent-javascript-robot",
  "description": "Simulación de un robot de entrega de paquetes",
  "version": "1.0.0",
  "main": "run.mjs",
  "dependencies": {
    "dijkstra.js": "^1.0.1",
    "random-item": "^1.0.0"
  },
  "license": "ISC"
}
```

Cuando ejecutas `npm install` sin especificar un paquete para instalar, NPM instalará las dependencias enumeradas en `package.json`. Cuando instalas un paquete específico que no está listado como una dependencia, NPM lo añadirá a `package.json`.

VERSIONES

Un archivo `package.json` lista tanto la versión del propio programa como las versiones de sus dependencias. Las versiones son una forma de manejar el hecho de que los paquetes evolucionan por separado, y el código escrito para funcionar con un paquete tal como existía en un momento dado puede no funcionar con una versión posterior y modificada del paquete.

NPM exige que sus paquetes sigan un esquema llamado *semantic versioning*, que codifica información sobre qué versiones son *compatibles* (no rompen la antigua interfaz) en el número de versión. Una versión semántica consiste en tres números, separados por puntos, como 2.3.0. Cada vez que se añade nueva funcionalidad, el número del medio debe incrementarse. Cada vez que se rompe la compatibilidad, de modo que el código existente que utiliza el paquete puede que no funcione con la nueva versión, el primer número debe incrementarse.

Un carácter de intercalación (^) delante del número de versión para una dependencia en package.json indica que se puede instalar cualquier versión compatible con el número dado. Por ejemplo, "^2.3.0" significaría que se permite cualquier versión mayor o igual a 2.3.0 y menor que 3.0.0.

El comando npm también se utiliza para publicar nuevos paquetes o nuevas versiones de paquetes. Si ejecutas npm publish en un directorio que tiene un archivo package.json, se publicará un paquete con el nombre y versión listados en el archivo JSON en el registro. Cualquiera puede publicar paquetes en NPM, aunque solo bajo un nombre de paquete que aún no esté en uso, ya que no sería bueno que personas aleatorias pudieran actualizar paquetes existentes. Este libro no profundizará más en los detalles del uso de NPM. Consulta <https://npmjs.org> para obtener más documentación y una forma de buscar paquetes.

EL MÓDULO DEL SISTEMA DE ARCHIVOS

Uno de los módulos integrados más utilizados en Node es el módulo `node:fs`, que significa *sistema de archivos*. Exporta funciones para trabajar con archivos y directorios.

Por ejemplo, la función llamada `readFile` lee un archivo y luego llama a una función de devolución de llamada con el contenido del archivo.

```
import {readFile} from "node:fs";
readFile("archivo.txt", "utf8", (error, texto) => {
  if (error) throw error;
  console.log("El archivo contiene:", texto);
});
```

El segundo argumento de `readFile` indica la *codificación de caracteres* utilizada para decodificar el archivo en una cadena. Existen varias formas en las que el texto puede ser codificado en datos binarios, pero la mayoría de los sistemas modernos utilizan UTF-8. Entonces, a menos que tengas razones para creer que se utiliza otra codificación, pasa `"utf8"` al leer un archivo de texto. Si no pasas una codificación, Node asumirá que estás interesado en los datos binarios y te dará un objeto `Buffer` en lugar de una cadena. Este es un objeto similar a un array que contiene números que representan los bytes (trozos de datos de 8 bits) en los archivos.

```
import {readFile} from "node:fs";
readFile("archivo.txt", (error, buffer) => {
  if (error) throw error;
  console.log("El archivo contenía", buffer.length, "bytes.",
    "El primer byte es:", buffer[0]);
});
```

Una función similar, `writeFile`, se utiliza para escribir un archivo en el disco.

```
import {writeFile} from "node:fs";
writeFile("graffiti.txt", "Node estuvo aquí", err => {
  if (err) console.log(`Error al escribir el archivo: ${err}`);
  else console.log("Archivo escrito.");
});
```

Aquí no fue necesario especificar la codificación: `writeFile` asumirá que cuando se le da una cadena para escribir, en lugar de un objeto `Buffer`, debe escribirla como texto utilizando su codificación de caracteres predeterminada, que es UTF-8.

El módulo `node:fs` contiene muchas otras funciones útiles: `readdir` te dará los archivos en un directorio como un array de cadenas, `stat` recuperará información sobre un archivo, `rename` cambiará el nombre de un archivo, `unlink` lo eliminará, entre otros. Consulta la documentación en <https://nodejs.org> para obtener detalles específicos.

La mayoría de estas funciones toman una función de devolución de llamada como último parámetro, a la que llaman ya sea con un error (el primer argumento) o con un resultado exitoso (el segundo). Como vimos en [Capítulo 11](#), hay desventajas en este estilo de programación, siendo la mayor que el manejo de errores se vuelve verboso y propenso a errores.

El módulo `node:fs/promises` exporta la mayoría de las mismas funciones que el antiguo módulo `node:fs`, pero utiliza promesas en lugar de funciones de devolución de llamada.

```
import {readFile} from "node:fs/promises";
readFile("file.txt", "utf8")
  .then(text => console.log("El archivo contiene:", text));
```

A veces no necesitas asincronía y simplemente te estorba. Muchas de las funciones en `node:fs` también tienen una variante síncrona, que tiene el mismo nombre con `Sync` agregado al final. Por ejemplo, la versión síncrona de `readFile` se llama `readFileSync`.

```
import {readFileSync} from "node:fs";
console.log("El archivo contiene:",
  readFileSync("file.txt", "utf8"));
```

Cabe destacar que mientras se realiza una operación síncrona de este tipo, tu programa se detiene por completo. Si debería estar respondiendo al usuario o a otras máquinas en la red, quedarse atrapado en una acción síncrona podría producir retrasos molestos.

EL MÓDULO HTTP

Otro módulo central se llama `node:http`. Proporciona funcionalidad para ejecutar un servidor HTTP.

Esto es todo lo que se necesita para iniciar un servidor HTTP:

```
import {createServer} from "node:http";
let server = createServer((solicitud, respuesta) => {
  respuesta.writeHead(200, {"Content-Type": "text/html"});
  respuesta.write(`
    <h1>¡Hola!</h1>
    <p>Pediste <code>${solicitud.url}</code></p>`);
  respuesta.end();
});
server.listen(8000);
console.log("¡Escuchando! (puerto 8000)");
```

Si ejecutas este script en tu propia máquina, puedes apuntar tu navegador web a <http://localhost:8000/hola> para hacer una solicitud a tu servidor. Responderá con una pequeña página HTML.

La función pasada como argumento a `createServer` se llama cada vez que un cliente se conecta al servidor. Los enlaces `solicitud` y `respuesta` son objetos que representan los datos de entrada y salida. El primero contiene información sobre la solicitud, como su propiedad `url`, que nos dice a qué URL se hizo la solicitud.

Así que, cuando abres esa página en tu navegador, envía una solicitud a tu propia computadora. Esto hace que la función del servidor se ejecute y envíe una respuesta, que luego puedes ver en el navegador.

Para enviar algo al cliente, llamas a métodos en el objeto `respuesta`. El primero, `writeHead`, escribirá los encabezados de respuesta (ver [Capítulo 18](#)). Le das el código de estado (200 para “OK” en este caso) y un objeto que contiene valores de encabezado. El ejemplo establece el encabezado `Content-Type` para informar al cliente que estaremos enviando de vuelta un documento HTML.

A continuación, el cuerpo real de la respuesta (el documento en sí) se envía con `response.write`. Se permite llamar a este método varias veces si deseas enviar la respuesta pieza por pieza, por ejemplo para transmitir datos al cliente a medida que estén disponibles. Por último, `response.end` señala el fin de la respuesta.

La llamada a `server.listen` hace que el servidor comience a esperar conexiones en el puerto 8000. Por eso debes conectarte a

localhost:8000 para comunicarte con este servidor, en lugar de simplemente a *localhost*, que usaría el puerto predeterminado 80.

Cuando ejecutas este script, el proceso se queda esperando. Cuando un script está escuchando eventos —en este caso, conexiones de red—, node no se cerrará automáticamente al llegar al final del script. Para cerrarlo, presiona CONTROL-C.

Un verdadero servidor web server usualmente hace más cosas que el ejemplo; examina el método de la solicitud (la propiedad `method`) para ver qué acción está intentando realizar el cliente y mira el URL de la solicitud para descubrir sobre qué recurso se está realizando esta acción. Veremos un servidor más avanzado [más adelante en este capítulo](#).

El módulo `node:http` también provee una función `request`, que se puede usar para hacer solicitudes HTTP. Sin embargo, es mucho más engorroso de usar que `fetch`, que vimos en [Capítulo 18](#).

Afortunadamente, `fetch` también está disponible en Node, como un enlace global. A menos que desees hacer algo muy específico, como procesar el documento de respuesta pieza por pieza a medida que llegan los datos a través de la red, recomiendo usar `fetch`.

FLUJOS

El objeto de respuesta al que el servidor HTTP podría escribir es un ejemplo de un objeto de *flujo de escritura*, que es un concepto ampliamente usado en Node. Estos objetos tienen un método `write` al que se puede pasar una cadena o un objeto `Buffer` para escribir algo en el flujo. Su método `end` cierra el flujo y opcionalmente toma un valor para escribir en el flujo antes de cerrarlo. Ambos métodos

también pueden recibir una devolución de llamada como argumento adicional, que se llamará cuando la escritura o el cierre hayan finalizado.

Es posible crear un flujo de escritura que apunte a un archivo con la función `createWriteStream` del módulo `node:fs`. Luego puedes usar el método `write` en el objeto resultante para escribir el archivo pieza por pieza, en lugar de hacerlo de una sola vez como con `writeFile`.

Los *flujos legibles* son un poco más complejos. El argumento `request` para la devolución de llamada del servidor HTTP es un flujo legible. Leer de un flujo se hace utilizando manejadores de eventos, en lugar de métodos.

Los objetos que emiten eventos en Node tienen un método llamado `on` que es similar al método `addEventListener` en el navegador. Le das un nombre de evento y luego una función, y registrará esa función para que se llame cada vez que ocurra el evento dado.

Los streams legibles tienen eventos `"data"` y `"end"`. El primero se dispara cada vez que llegan datos, y el segundo se llama cuando el flujo llega a su fin. Este modelo es más adecuado para datos de *streaming* que pueden procesarse de inmediato, incluso cuando todo el documento aún no está disponible. Un archivo se puede leer como un flujo legible utilizando la función `createReadStream` de `node:fs`.

Este código crea un servidor que lee los cuerpos de las solicitudes y los reenvía al cliente como texto en mayúsculas:

```
import {createServer} from "node:http";
createServer((solicitud, respuesta) => {
  respuesta.writeHead(200, {"Content-Type": "text/plain"});
  solicitud.on("data", fragmento =>
    respuesta.write(fragmento.toString().toUpperCase()));
  solicitud.on("end", () => respuesta.end());
}).listen(8000);
```

El valor chunk pasado al controlador de datos será un Buffer binario. Podemos convertir esto a una cadena decodificándolo como caracteres codificados en UTF-8 con su método `toString`.

El siguiente fragmento de código, cuando se ejecuta con el servidor de mayúsculas activo, enviará una solicitud a ese servidor y escribirá la respuesta que recibe:

```
fetch("http://localhost:8000/", {
  method: "POST",
  body: "Hola servidor"
}).then(resp => resp.text()).then(console.log);
// → HOLA SERVIDOR
```

UN SERVIDOR DE ARCHIVOS

Combina nuestro nuevo conocimiento sobre los servidores HTTP y el trabajo con el sistema de archivos para crear un puente entre ambos: un servidor HTTP que permite el acceso remoto a un sistema de archivos. Este tipo de servidor tiene todo tipo de usos, como permitir que las aplicaciones web almacenen y compartan datos, o dar acceso compartido a un grupo de personas a un montón de archivos.

Cuando tratamos los archivos como recursos de HTTP, los métodos HTTP GET, PUT y DELETE se pueden usar para leer, escribir y

eliminar los archivos, respectivamente. Interpretaremos la ruta en la solicitud como la ruta del archivo al que se refiere la solicitud.

Probablemente no queramos compartir todo nuestro sistema de archivos, por lo que interpretaremos estas rutas como comenzando en el directorio de trabajo del servidor, que es el directorio en el que se inició. Si ejecuté el servidor desde `/tmp/public/` (o `C:\tmp\public\` en Windows), entonces una solicitud para `/file.txt` debería referirse a `/tmp/public/file.txt` (o `C:\tmp\public\file.txt`).

Construiremos el programa paso a paso, utilizando un objeto llamado `methods` para almacenar las funciones que manejan los diferentes métodos HTTP. Los controladores de métodos son funciones `async` que reciben el objeto de solicitud como argumento y devuelven una promesa que se resuelve a un objeto que describe la respuesta.

```
import {createServer} from "node:http";

const methods = Object.create(null);

createServer((request, response) => {
  let handler = methods[request.method] || notAllowed;
  handler(request).catch(error => {
    if (error.status !== null) return error;
    return {body: String(error), status: 500};
  }).then(({body, status = 200, type = "text/plain"}) => {
    response.writeHead(status, {"Content-Type": type});
    if (body && body.pipe) body.pipe(response);
    else response.end(body);
  });
}).listen(8000);

async function notAllowed(request) {
```

```
return {  
  status: 405,  
  body: `Método ${request.method} no permitido.`  
};  
}
```

Esto inicia un servidor que simplemente devuelve respuestas de error 405, que es el código utilizado para indicar que el servidor se niega a manejar un método determinado.

Cuando la promesa de un controlador de solicitud es rechazada, la llamada a `catch` traduce el error en un objeto de respuesta, si aún no lo es, para que el servidor pueda enviar una respuesta de error para informar al cliente que no pudo manejar la solicitud.

El campo `status` de la descripción de la respuesta puede omitirse, en cuyo caso se establece en 200 (OK) por defecto. El tipo de contenido, en la propiedad `type`, también puede omitirse, en cuyo caso se asume que la respuesta es texto plano.

Cuando el valor de `body` es un `readable stream`, este tendrá un método `pipe` que se utiliza para reenviar todo el contenido de un flujo de lectura a un `writable stream`. Si no es así, se asume que es `null` (sin cuerpo), una cadena o un búfer, y se pasa directamente al método `end` del `response`.

Para determinar qué ruta de archivo corresponde a una URL de solicitud, la función `urlPath` utiliza la clase integrada `URL` (que también existe en el navegador) para analizar la URL. Este constructor espera una URL completa, no solo la parte que comienza con la barra diagonal que obtenemos de `request.url`, por lo que le proporcionamos un nombre de dominio falso para completar.

Extrae su ruta, que será algo como `"/archivo.txt"`, la decodifica para eliminar los códigos de escape estilo `%20`, y la resuelve en relación con el directorio de trabajo del programa.

```
import {parse} from "node:url";
import {resolve, sep} from "node:path";

const baseDirectory = process.cwd();

function urlPath(url) {
  let {pathname} = new URL(url, "http://d");
  let path = resolve(decodeURIComponent(pathname).slice(1));
  if (path !== baseDirectory &&
      !path.startsWith(baseDirectory + sep)) {
    throw {status: 403, body: "Prohibido"};
  }
  return path;
}
```

Tan pronto como configuras un programa para aceptar solicitudes de red, debes empezar a preocuparte por la seguridad. En este caso, si no tenemos cuidado, es probable que terminemos exponiendo accidentalmente todo nuestro sistema de archivos a la red.

Las rutas de archivos son cadenas en Node. Para mapear dicha cadena a un archivo real, hay una cantidad no trivial de interpretación en juego. Las rutas pueden, por ejemplo, incluir `../` para hacer referencia a un directorio padre. Así que una fuente obvia de problemas serían las solicitudes de rutas como `../archivo_secreto`.

Para evitar tales problemas, `urlPath` utiliza la función `resolve` del módulo `node:path`, que resuelve rutas relativas. Luego verifica que el resultado esté *debajo* del directorio de trabajo. La función

`process.cwd` (donde `cwd` significa “directorio de trabajo actual”) se puede usar para encontrar este directorio de trabajo. El vínculo `sep` del paquete `node:path` es el separador de ruta del sistema: una barra invertida en Windows y una barra diagonal en la mayoría de otros sistemas. Cuando la ruta no comienza con el directorio base, la función arroja un objeto de respuesta de error, usando el código de estado HTTP que indica que el acceso al recurso está prohibido.

Configuraremos el método GET para devolver una lista de archivos al leer un directorio y para devolver el contenido del archivo al leer un archivo regular.

Una pregunta complicada es qué tipo de encabezado `Content-Type` debemos establecer al devolver el contenido de un archivo. Dado que estos archivos podrían ser cualquier cosa, nuestro servidor no puede simplemente devolver el mismo tipo de contenido para todos ellos. npm puede ayudarnos nuevamente aquí. El paquete `mime-types` (los indicadores de tipo de contenido como `text/plain` también se llaman *tipos MIME*) conoce el tipo correcto para una gran cantidad de extensiones de archivo.

El siguiente comando de npm, en el directorio donde reside el script del servidor, instala una versión específica de `mime`:

```
$ npm install mime-types@2.1.0
```

Cuando un archivo solicitado no existe, el código de estado HTTP correcto a devolver es 404. Utilizaremos la función `stat`, que busca información sobre un archivo, para averiguar tanto si el archivo existe como si es un directorio.

```

import {createReadStream} from "node:fs";
import {stat, readdir} from "node:fs/promises";
import {lookup} from "mime-types";

methods.GET = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code !== "ENOENT") throw error;
    else return {status: 404, body: "Archivo no encontrado"};
  }
  if (stats.isDirectory()) {
    return {body: (await readdir(path)).join("\n")};
  } else {
    return {body: createReadStream(path),
            type: lookup(path)};
  }
};

```

Debido a que debe acceder al disco y por lo tanto podría llevar algún tiempo, `stat` es asíncrono. Dado que estamos utilizando promesas en lugar del estilo de devolución de llamada, debe ser importado desde `node:fs/promises` en lugar de directamente desde `node:fs`.

Cuando el archivo no existe, `stat` lanzará un objeto de error con una propiedad `code` de `"ENOENT"`. Estos códigos algo oscuros, inspirados en Unix, son la forma en que se reconocen los tipos de error en Node.

El objeto `stats` devuelto por `stat` nos indica varias cosas sobre un archivo, como su tamaño (propiedad `size`) y su fecha de modificación (`mtime`). Aquí nos interesa saber si es un directorio o un archivo regular, lo cual nos dice el método `isDirectory`.

Usamos `readdir` para leer la matriz de archivos en un directorio y devolverla al cliente. Para archivos normales, creamos un flujo de lectura con `createReadStream` y lo devolvemos como cuerpo, junto con el tipo de contenido que nos proporciona el paquete `mime` para el nombre del archivo.

El código para manejar las solicitudes `DELETE` es ligeramente más sencillo.

```
import {rmdir, unlink} from "node:fs/promises";

methods.DELETE = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code !== "ENOENT") throw error;
    else return {status: 204};
  }
  if (stats.isDirectory()) await rmdir(path);
  else await unlink(path);
  return {status: 204};
};
```

Cuando una respuesta HTTP no contiene datos, se puede usar el código de estado 204 (“sin contenido”) para indicarlo. Dado que la respuesta a la eliminación no necesita transmitir ninguna información más allá de si la operación tuvo éxito, es sensato devolver eso aquí.

Es posible que te preguntes por qué intentar eliminar un archivo inexistente devuelve un código de estado de éxito en lugar de un error. Cuando el archivo que se está eliminando no está presente, se podría decir que el objetivo de la solicitud ya se ha cumplido. El

estándar HTTP nos anima a hacer solicitudes *idempotentes*, lo que significa que hacer la misma solicitud varias veces produce el mismo resultado que hacerla una vez. De cierta manera, si intentas eliminar algo que ya no está, el efecto que intentabas lograr se ha alcanzado: la cosa ya no está allí.

Este es el manejador para las solicitudes PUT:

```
import {createWriteStream} from "node:fs";

function pipeStream(from, to) {
  return new Promise((resolve, reject) => {
    from.on("error", reject);
    to.on("error", reject);
    to.on("finish", resolve);
    from.pipe(to);
  });
}

methods.PUT = async function(request) {
  let path = urlPath(request.url);
  await pipeStream(request, createWriteStream(path));
  return {status: 204};
};
```

Esta vez no necesitamos verificar si el archivo existe; si lo hace, simplemente lo sobrescribiremos. Nuevamente usamos pipe para mover datos de un flujo legible a uno escribible, en este caso del request al archivo. Pero como pipe no está diseñado para devolver una promesa, debemos escribir un contenedor, pipeStream, que cree una promesa alrededor del resultado de llamar a pipe.

Cuando algo sale mal al abrir el archivo, createWriteStream seguirá devolviendo un flujo, pero ese flujo lanzará un evento de "error". El flujo del request también puede fallar, por ejemplo si la red falla. Por lo tanto, conectamos los eventos de "error" de

ambos flujos para rechazar la promesa. Cuando pipe haya terminado, cerrará el flujo de salida, lo que hará que lance un evento de "finalización". En ese momento podemos resolver la promesa con éxito (devolviendo nada).

El script completo del servidor está disponible en https://eloquentjavascript.net/code/file_server.mjs. Puedes descargarlo y, después de instalar sus dependencias, ejecutarlo con Node para iniciar tu propio servidor de archivos. Y, por supuesto, puedes modificarlo y ampliarlo para resolver los ejercicios de este capítulo o para experimentar.

La herramienta de línea de comandos `curl`, ampliamente disponible en sistemas Unix (como macOS y Linux), se puede utilizar para hacer solicitudes HTTP. La siguiente sesión prueba brevemente nuestro servidor. La opción `-X` se usa para establecer el método de la solicitud, y `-d` se utiliza para incluir un cuerpo de solicitud.

```
$ curl http://localhost:8000/file.txt
Archivo no encontrado
$ curl -X PUT -d CONTENIDO http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
CONTENIDO
$ curl -X DELETE http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
Archivo no encontrado
```

La primera solicitud para `file.txt` falla ya que el archivo aún no existe. La solicitud `PUT` crea el archivo y, voilà, la siguiente solicitud lo recupera con éxito. Después de eliminarlo con una solicitud `DELETE`, el archivo vuelve a estar ausente.

RESUMEN

Node es un sistema pequeño interesante que nos permite ejecutar JavaScript en un contexto no de navegador. Originalmente fue diseñado para tareas de red para desempeñar el papel de un *nodo* en una red. Sin embargo, se presta para todo tipo de tareas de script, y si disfrutas escribir JavaScript, automatizar tareas con Node funciona bien.

NPM proporciona paquetes para todo lo que puedas imaginar (y varias cosas que probablemente nunca se te ocurrirían), y te permite descargar e instalar esos paquetes con el programa `npm`. Node viene con varios módulos integrados, incluido el módulo `node:fs` para trabajar con el sistema de archivos y el módulo `node:http` para ejecutar servidores HTTP. Todo el input y output en Node se hace de forma asíncrona, a menos que uses explícitamente una variante síncrona de una función, como `readFileSync`. Originalmente, Node usaba devoluciones de llamada para funcionalidades asíncronas, pero el paquete `node:fs/promises` proporciona una interfaz basada en promesas para el sistema de archivos.

EJERCICIOS

HERRAMIENTA DE BÚSQUEDA

En los sistemas Unix, existe una herramienta de línea de comandos llamada `grep` que se puede utilizar para buscar rápidamente archivos según una expresión regular.

Escribe un script de Node que se pueda ejecutar desde la línea de comandos y funcione de manera similar a `grep`. Trata el primer argumento de la línea de comandos como una expresión regular y trata cualquier argumento adicional como archivos a buscar. Debería

mostrar los nombres de los archivos cuyo contenido coincide con la expresión regular.

Una vez que eso funcione, extiéndelo para que cuando uno de los argumentos sea un directorio, busque en todos los archivos de ese directorio y sus subdirectorios.

Utiliza funciones asíncronas o síncronas del sistema de archivos según consideres adecuado. Configurar las cosas para que se soliciten múltiples acciones asíncronas al mismo tiempo podría acelerar un poco las cosas, pero no demasiado, ya que la mayoría de los sistemas de archivos solo pueden leer una cosa a la vez.

CREACIÓN DE DIRECTORIOS

Aunque el método DELETE en nuestro servidor de archivos es capaz de eliminar directorios (usando `rmdir`), actualmente el servidor no proporciona ninguna forma de *crear* un directorio.

Añade soporte para el método MKCOL (“make collection”), que debería crear un directorio llamando a `mkdir` desde el módulo `node:fs`. MKCOL no es un método HTTP ampliamente utilizado, pero sí existe con este mismo propósito en el estándar *WebDAV*, el cual especifica un conjunto de convenciones sobre HTTP que lo hacen adecuado para crear documentos.

UN ESPACIO PÚBLICO EN LA WEB

Dado que el servidor de archivos sirve cualquier tipo de archivo e incluso incluye la cabecera `Content-Type` correcta, puedes usarlo para servir un sitio web. Dado que permite a todos eliminar y

reemplazar archivos, sería un tipo interesante de sitio web: uno que puede ser modificado, mejorado y vandalizado por todos aquellos que se tomen el tiempo de hacer la solicitud HTTP adecuada.

Escribe una página HTML básica que incluya un archivo JavaScript sencillo. Coloca los archivos en un directorio servido por el servidor de archivos y ábrelos en tu navegador.

Luego, como ejercicio avanzado o incluso como un proyecto de fin de semana, combina todo el conocimiento que has adquirido de este libro para construir una interfaz más amigable para modificar el sitio web—desde *dentro* del sitio web.

Utiliza un formulario HTML para editar el contenido de los archivos que conforman el sitio web, permitiendo al usuario actualizarlos en el servidor mediante solicitudes HTTP, como se describe en [Capítulo 18](#).

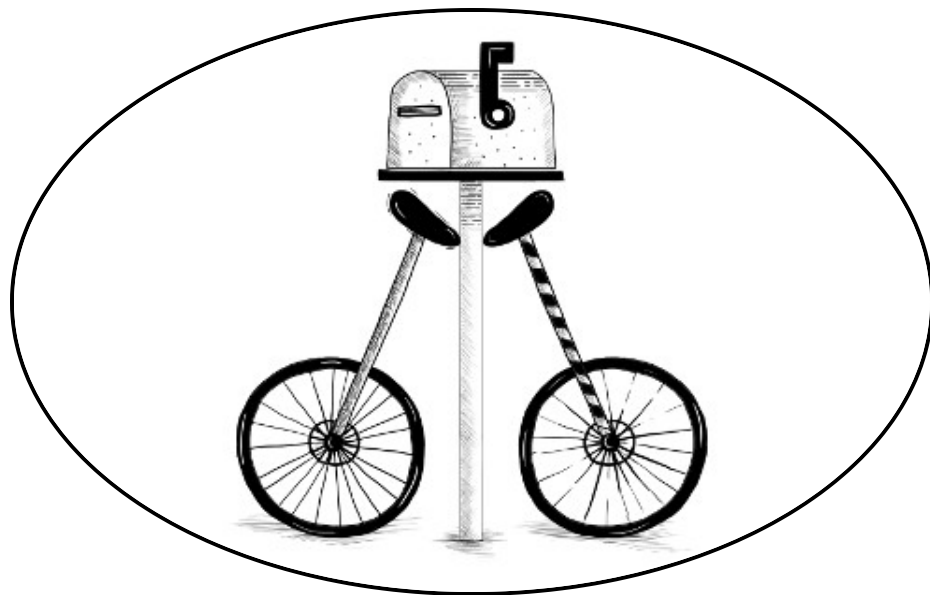
Comienza permitiendo que solo un archivo sea editable. Luego haz que el usuario pueda seleccionar qué archivo editar. Aprovecha el hecho de que nuestro servidor de archivos devuelve listas de archivos al leer un directorio.

No trabajes directamente en el código expuesto por el servidor de archivos ya que si cometes un error, es probable que dañes los archivos allí. En su lugar, mantén tu trabajo fuera del directorio accesible al público y cópialo allí al hacer pruebas.

PROYECTO: SITIO WEB DE INTERCAMBIO DE HABILIDADES

“Si tienes conocimiento, permite que otros enciendan sus velas en él.”

—Margaret Fuller



Una reunión de intercambio de habilidades es un evento en el que personas con un interés compartido se reúnen y dan pequeñas presentaciones informales sobre cosas que saben. En una reunión de intercambio de habilidades de jardinería, alguien podría explicar cómo cultivar apio. O en un grupo de intercambio de habilidades de programación, podrías pasar y contarles a la gente sobre Node.js.

En este último capítulo del proyecto, nuestro objetivo es configurar un sitio web para gestionar las charlas impartidas en una reunión de

intercambio de habilidades. Imagina un pequeño grupo de personas que se reúnen regularmente en la oficina de uno de los miembros para hablar sobre monociclos. El organizador anterior de las reuniones se mudó a otra ciudad y nadie se ofreció a asumir esta tarea. Queremos un sistema que permita a los participantes proponer y discutir charlas entre ellos, sin un organizador activo.

El código completo del proyecto se puede descargar desde <https://eloquentjavascript.net/code/skillsharing.zip>.

DISEÑO

Este proyecto tiene una parte de *servidor*, escrita para Node.js, y una parte de *cliente*, escrita para el navegador. El servidor almacena los datos del sistema y los proporciona al cliente. También sirve los archivos que implementan el sistema del lado del cliente.

El servidor mantiene la lista de charlas propuestas para la próxima reunión, y el cliente muestra esta lista. Cada charla tiene un nombre de presentador, un título, un resumen y una matriz de comentarios asociados. El cliente permite a los usuarios proponer nuevas charlas (agregándolas a la lista), eliminar charlas y comentar en charlas existentes. Cada vez que el usuario realiza un cambio de este tipo, el cliente realiza una solicitud HTTP para informar al servidor al respecto.

Skill Sharing

Your name:

Unituning

by Jamal

Modifying your cycle for extra style

Iman: *Will you talk about raising a cycle?*

Jamal: *Definitely*

Iman: *I'll be there*

Submit a talk

Title:

Summary:

La aplicación se configurará para mostrar una vista *en vivo* de las charlas propuestas actuales y sus comentarios. Cada vez que alguien, en algún lugar, envíe una nueva charla o agregue un comentario, todas las personas que tengan la página abierta en sus navegadores deberían ver el cambio de inmediato. Esto plantea un desafío—no hay forma de que un servidor web abra una conexión a un cliente, ni hay una buena forma de saber qué clientes están viendo actualmente un sitio web dado.

Una solución común a este problema se llama *long polling*, que resulta ser una de las motivaciones del diseño de Node.

LONG POLLING

Para poder notificar inmediatamente a un cliente que algo ha cambiado, necesitamos una conexión con ese cliente. Dado que los navegadores web tradicionalmente no aceptan conexiones y los clientes a menudo están detrás de routers que bloquearían tales conexiones de todos modos, no es práctico que sea el servidor quien inicie esta conexión.

Podemos hacer que el cliente abra la conexión y la mantenga activa para que el servidor pueda usarla para enviar información cuando sea necesario.

Sin embargo, una solicitud HTTP permite solo un flujo simple de información: el cliente envía una solicitud, el servidor responde una sola vez, y eso es todo. Existe una tecnología llamada *WebSockets* que permite abrir conexiones para el intercambio arbitrario de datos. Pero usarlas adecuadamente es algo complicado.

En este capítulo, utilizamos una técnica más sencilla—long polling—donde los clientes preguntan continuamente al servidor por nueva información mediante solicitudes HTTP regulares, y el servidor retiene su respuesta cuando no tiene nada nuevo que informar.

Mientras el cliente se asegure de tener una solicitud de sondeo abierta constantemente, recibirá información del servidor rápidamente cuando esté disponible. Por ejemplo, si Fatma tiene nuestra aplicación de intercambio de habilidades abierta en su navegador, ese navegador habrá solicitado actualizaciones y estará esperando una respuesta a esa solicitud. Cuando Iman envía una charla sobre “Extreme Downhill Unicycling”, el servidor notará que Fatma está esperando actualizaciones y enviará una respuesta que contiene la nueva charla a su solicitud pendiente. El navegador de

Fatma recibirá los datos y actualizará la pantalla para mostrar la charla.

Para evitar que las conexiones se agoten por tiempo (se aborten debido a una falta de actividad), las técnicas de long polling suelen establecer un tiempo máximo para cada solicitud, tras el cual el servidor responderá de todos modos, aunque no tenga nada que informar. Entonces, el cliente puede iniciar una nueva solicitud. Reiniciar periódicamente la solicitud también hace que la técnica sea más robusta, permitiendo a los clientes recuperarse de fallos temporales de conexión o problemas de servidor.

Un servidor ocupado que utiliza long polling puede tener miles de solicitudes en espera, y por lo tanto conexiones TCP abiertas. Node, que facilita la gestión de muchas conexiones sin crear un hilo de control separado para cada una, es ideal para este tipo de sistema.

INTERFAZ HTTP

Antes de comenzar a diseñar el servidor o el cliente, pensemos en el punto donde se conectan: la interfaz HTTP a través de la cual se comunican.

Utilizaremos JSON como formato de nuestro cuerpo de solicitud y respuesta. Al igual que en el servidor de archivos del [Capítulo 20](#), intentaremos hacer un buen uso de los métodos y cabeceras HTTP. La interfaz se centra en la ruta `/talks`. Las rutas que no comienzan con `/talks` se utilizarán para servir archivos estáticos—el código HTML y JavaScript para el sistema del lado del cliente.

Una solicitud GET a `/talks` devuelve un documento JSON como este:

```
[{"title": "Unituning",  
  "presenter": "Jamal",  
  "summary": "Modificando tu bicicleta para darle más estilo",  
  "comments": []}]
```

Crear una nueva charla se hace haciendo una solicitud PUT a una URL como `/talks/Unituning`, donde la parte después de la segunda barra es el título de la charla. El cuerpo de la solicitud PUT debe contener un objeto JSON que tenga propiedades `presenter` y `summary`.

Dado que los títulos de las charlas pueden contener espacios y otros caracteres que normalmente no aparecen en una URL, las cadenas de título deben ser codificadas con la función `encodeURIComponent` al construir una URL de ese tipo.

```
console.log("/talks/" + encodeURIComponent("Cómo hacer el  
caballito"));  
// → /talks/Cómo%20hacer%20el%20caballito
```

Una solicitud para crear una charla sobre hacer el caballito podría ser algo así:

```
PUT /talks/Cómo%20hacer%20el%20caballito HTTP/1.1  
Content-Type: application/json  
Content-Length: 92
```

```
{"presenter": "Maureen",  
  "summary": "Permanecer quieto sobre un monociclo"}
```

Estas URLs también admiten solicitudes GET para recuperar la representación JSON de una charla y solicitudes DELETE para eliminar una charla.

Agregar un comentario a una charla se hace con una solicitud POST a una URL como `/talks/Unituning/comments`, con un cuerpo JSON que tiene propiedades `author` y `message`.

```
POST /talks/Unituning/comments HTTP/1.1
Content-Type: application/json
Content-Length: 72
```

```
{"author": "Iman",
 "message": "¿Vas a hablar sobre cómo levantar una bicicleta?"}
```

Para soportar encuestas prolongadas, las solicitudes GET a `/talks` pueden incluir encabezados adicionales que informen al servidor para retrasar la respuesta si no hay nueva información disponible. Usaremos un par de encabezados normalmente destinados a gestionar el almacenamiento en caché: `ETag` y `If-None-Match`.

Los servidores pueden incluir un encabezado `ETag` (“etiqueta de entidad”) en una respuesta. Su valor es una cadena que identifica la versión actual del recurso. Los clientes, al solicitar posteriormente ese recurso de nuevo, pueden hacer una *solicitud condicional* incluyendo un encabezado `If-None-Match` cuyo valor contenga esa misma cadena. Si el recurso no ha cambiado, el servidor responderá con el código de estado 304, que significa “no modificado”, indicando al cliente que su versión en caché sigue siendo actual. Cuando la etiqueta no coincide, el servidor responde como de costumbre.

Necesitamos algo como esto, donde el cliente puede decirle al servidor qué versión de la lista de charlas tiene, y el servidor responde solo cuando esa lista ha cambiado. Pero en lugar de devolver inmediatamente una respuesta 304, el servidor debería demorar la respuesta y devolverla solo cuando haya algo nuevo disponible o haya transcurrido una cantidad de tiempo determinada. Para distinguir las solicitudes de encuestas prolongadas de las solicitudes condicionales normales, les damos otro encabezado, `Prefer: wait=90`, que le indica al servidor que el cliente está dispuesto a esperar hasta 90 segundos por la respuesta. El servidor mantendrá un número de versión que actualiza cada vez que cambian las charlas y lo utilizará como valor `ETag`. Los clientes pueden hacer solicitudes como esta para ser notificados cuando las charlas cambien:

```
GET /talks HTTP/1.1
If-None-Match: "4"
Prefer: wait=90

(pasa el tiempo)

HTTP/1.1 200 OK
Content-Type: application/json
ETag: "5"
Content-Length: 295

[....]
```

El protocolo descrito aquí no realiza ningún control de acceso. Cualquiera puede comentar, modificar charlas e incluso eliminarlas. (Dado que Internet está lleno de matones, poner un sistema en línea sin una protección adicional probablemente no terminaría bien).

EL SERVIDOR

Comencemos construyendo la parte del programa del lado del servidor. El código en esta sección se ejecuta en Node.js.

ENRUTAMIENTO

Nuestro servidor utilizará `createServer` de Node para iniciar un servidor HTTP. En la función que maneja una nueva solicitud, debemos distinguir entre los diferentes tipos de solicitudes (como se determina por el método y la ruta) que soportamos. Esto se puede hacer con una larga cadena de declaraciones `if`, pero hay una manera más elegante.

Un *enrutador* es un componente que ayuda a despachar una solicitud a la función que puede manejarla. Puedes indicarle al enrutador, por ejemplo, que las solicitudes PUT con una ruta que coincida con la expresión regular `/^\/talks\/([^\/]*)$/` (`/talks/` seguido de un título de charla) pueden ser manejadas por una función dada. Además, puede ayudar a extraer las partes significativas de la ruta (en este caso el título de la charla), envueltas en paréntesis en la expresión regular, y pasarlas a la función manejadora.

Hay varios paquetes de enrutadores buenos en NPM, pero aquí escribiremos uno nosotros mismos para ilustrar el principio.

Este es `router.mjs`, que luego importaremos desde nuestro módulo del servidor:

```
export class Router {  
  constructor() {
```

```

    this.routes = [];
  }
  add(method, url, handler) {
    this.routes.push({method, url, handler});
  }
  async resolve(request, context) {
    let {pathname} = new URL(request.url, "http://d");
    for (let {method, url, handler} of this.routes) {
      let match = url.exec(pathname);
      if (!match || request.method !== method) continue;
      let parts = match.slice(1).map(decodeURIComponent);
      return handler(context, ...parts, request);
    }
  }
}

```

El módulo exporta la clase `Router`. Un objeto de enrutador te permite registrar manejadores para métodos específicos y patrones de URL con su método `add`. Cuando una solicitud se resuelve con el método `resolve`, el enrutador llama al manejador cuyo método y URL coinciden con la solicitud y devuelve su resultado.

Las funciones manejadoras se llaman con el valor `context` dado a `resolve`. Utilizaremos esto para darles acceso al estado de nuestro servidor. Además, reciben las cadenas coincidentes para cualquier grupo que hayan definido en su expresión regular, y el objeto de solicitud. Las cadenas deben ser decodificadas de la URL ya que la URL cruda puede contener códigos estilo `%20`.

SIRVIENDO ARCHIVOS

Cuando una solicitud no coincide con ninguno de los tipos de solicitud definidos en nuestro enrutador, el servidor debe interpretarlo como una solicitud de un archivo en el directorio `public`. Sería posible usar el servidor de archivos definido en

[Capítulo 20](#) para servir dichos archivos, pero ni necesitamos ni queremos admitir solicitudes PUT y DELETE en archivos, y nos gustaría tener funciones avanzadas como el soporte para almacenamiento en caché. Así que usemos en cambio un servidor de archivos estático sólido y bien probado de NPM.

Opté por `serve-static`. Este no es el único servidor de este tipo en NPM, pero funciona bien y se ajusta a nuestros propósitos. El paquete `serve-static` exporta una función que puede ser llamada con un directorio raíz para producir una función manipuladora de solicitudes. La función manipuladora acepta los argumentos `request` y `response` proporcionados por el servidor de `"node:http"`, y un tercer argumento, una función que se llamará si ningún archivo coincide con la solicitud. Queremos que nuestro servidor primero compruebe las solicitudes que deberíamos manejar de manera especial, según lo definido en el enrutador, por lo que lo envolvemos en otra función.

```
import {createServer} from "node:http";
import serveStatic from "serve-static";

function notFound(request, response) {
  response.writeHead(404, "Not found");
  response.end("<h1>Not found</h1>");
}

class SkillShareServer {
  constructor(talks) {
    this.talks = talks;
    this.version = 0;
    this.waiting = [];

    let fileServer = serveStatic("./public");
    this.server = createServer((request, response) => {
      serveFromRouter(this, request, response, () => {
```



```

        fileServer(request, response,
                    () => notFound(request, response));
    });
});
}
start(port) {
    this.server.listen(port);
}
stop() {
    this.server.close();
}
}

```

La función `serveFromRouter` tiene la misma interfaz que `fileServer`, tomando los argumentos (`request`, `response`, `next`). Esto nos permite “encadenar” varios manipuladores de solicitudes, permitiendo que cada uno maneje la solicitud o pase la responsabilidad de eso al siguiente manejador. El manejador final, `notFound`, simplemente responde con un error de “no encontrado”.

Nuestra función `serveFromRouter` utiliza una convención similar a la del servidor de archivos del [capítulo anterior](#) para las respuestas: los manejadores en el enrutador devuelven promesas que se resuelven en objetos que describen la respuesta.

```

import {Router} from "../router.mjs";

const router = new Router();
const defaultHeaders = {"Content-Type": "text/plain"};

async function serveFromRouter(server, request,
                                response, next) {
    let resolved = await router.resolve(request, server)
    .catch(error => {
        if (error.status != null) return error;
        return {body: String(error), status: 500};
    });
}

```

```

    if (!resolved) return next();
    let {body, status = 200, headers = defaultHeaders} =
      await resolved;
    response.writeHead(status, headers);
    response.end(body);
  }

```

CHARLAS COMO RECURSOS

Las charlas que se han propuesto se almacenan en la propiedad `talks` del servidor, un objeto cuyas propiedades son los títulos de las charlas. Agregaremos algunos controladores a nuestro enrutador que expongan estos como recursos HTTP bajo `/charlas/[título]`.

El controlador para las solicitudes que GET una sola charla debe buscar la charla y responder ya sea con los datos JSON de la charla o con una respuesta de error 404.

```

const talkPath = /^\/charlas\/([^\/]*)$/;

router.add("GET", talkPath, async (server, title) => {
  if (Object.hasOwn(server.talks, title)) {
    return {body: JSON.stringify(server.talks[title]),
      headers: {"Content-Type": "application/json"}};
  } else {
    return {status: 404, body: `No se encontró la charla
    '${title}'`};
  }
});

```

Eliminar una charla se hace eliminándola del objeto `talks`.

```

router.add("DELETE", talkPath, async (server, title) => {
  if (Object.hasOwn(server.talks, title)) {
    delete server.talks[title];
    server.updated();
  }
});

```

```
    }  
    return {status: 204};  
  });
```

El método `updated`, que definiremos [más adelante](#), notifica a las solicitudes de espera larga sobre el cambio.

Un controlador que necesita leer cuerpos de solicitud es el controlador `PUT`, que se utiliza para crear nuevas charlas. Debe verificar si los datos que se le proporcionaron tienen propiedades `presentador` y `resumen`, que son cadenas de texto. Cualquier dato que provenga de fuera del sistema podría ser un sinsentido y no queremos corromper nuestro modelo de datos interno o fallar cuando lleguen solicitudes incorrectas.

Si los datos parecen válidos, el controlador almacena un objeto que representa la nueva charla en el objeto `talks`, posiblemente sobrescribiendo una charla existente con este título, y nuevamente llama a `updated`.

Para leer el cuerpo del flujo de solicitud, utilizaremos la función `json` de `"node:stream/consumers"`, que recopila los datos en el flujo y luego los analiza como JSON. Hay exportaciones similares llamadas `text` (para leer el contenido como una cadena) y `buffer` (para leerlo como datos binarios) en este paquete. Dado que `json` es un nombre genérico, la importación lo renombra a `readJSON` para evitar confusiones.

```
import {json as readJSON} from "node:stream/consumers"  
  
router.add("PUT", talkPath,  
  async (server, title, request) => {  
    let talk = await readJSON(request);
```

```

    if (!talk ||
        typeof talk.presenter !== "string" ||
        typeof talk.summary !== "string") {
      return {status: 400, body: "Datos de charla incorrectos"};
    }
    server.talks[title] = {
      title,
      presenter: talk.presenter,
      summary: talk.summary,
      comments: []
    };
    server.updated();
    return {status: 204};
  });
  ```Agregar un ((comentario)) a una ((charla)) funciona de manera
 similar. Usamos `readJSON` para obtener el contenido de la
 solicitud, validamos los datos resultantes y los almacenamos como
 un comentario cuando parecen válidos.

  ```{includeCode: ">code/skillsharing/skillsharing_server.mjs"}
  router.add("POST", /^\/talks\/([^\/]+)\/comments$/,
    async (server, title, request) => {
    let comment = await readJSON(request);
    if (!comment ||
        typeof comment.author !== "string" ||
        typeof comment.message !== "string") {
      return {status: 400, body: "Datos de comentario incorrectos"};
    } else if (Object.hasOwn(server.talks, title)) {
      server.talks[title].comments.push(comment);
      server.updated();
      return {status: 204};
    } else {
      return {status: 404, body: `No se encontró la charla
      '${title}'`};
    }
  });

```

Intentar agregar un comentario a una charla inexistente devuelve un error 404.

SOPORTE PARA LARGA ESPERA

El aspecto más interesante del servidor es la parte que maneja la larga espera. Cuando llega una solicitud GET para /charlas, puede ser una solicitud regular o una solicitud de larga espera.

Habrá varios lugares en los que debamos enviar una matriz de charlas al cliente, por lo que primero definimos un método auxiliar que construya dicha matriz e incluya un encabezado ETag en la respuesta.

```
SkillShareServer.prototype.talkResponse = function() {
  let talks = Object.keys(this.talks)
    .map(title => this.talks[title]);
  return {
    body: JSON.stringify(talks),
    headers: {"Content-Type": "application/json",
              "ETag": `"$${this.version}"`,
              "Cache-Control": "no-store"}
  };
};
```

El controlador en sí mismo necesita examinar los encabezados de la solicitud para ver si están presentes los encabezados If-None-Match y Prefer. Node almacena los encabezados, cuyos nombres se especifican como insensibles a mayúsculas y minúsculas, bajo sus nombres en minúsculas.

```
router.add("GET", /^\/talks$/, async (server, request) => {
  let tag = /"(.*)""/.exec(request.headers["if-none-match"]);
  let wait = /\bwait=(\d+)/.exec(request.headers["prefer"]);
  if (!tag || tag[1] !== server.version) {
    return server.talkResponse();
  } else if (!wait) {
    return {status: 304};
  } else {
    return server.waitForChanges(Number(wait[1]));
  }
});
```

```
    }  
  });
```

Si no se proporcionó ninguna etiqueta o se proporcionó una etiqueta que no coincide con la versión actual del servidor, el controlador responde con la lista de charlas. Si la solicitud es condicional y las charlas no han cambiado, consultamos el encabezado `Prefer` para ver si debemos retrasar la respuesta o responder de inmediato.

Las funciones de devolución de llamada para solicitudes retardadas se almacenan en la matriz `waiting` del servidor para que puedan ser notificadas cuando ocurra algo. El método `waitForChanges` también establece inmediatamente un temporizador para responder con un estado `304` cuando la solicitud haya esperado el tiempo suficiente.

```
SkillShareServer.prototype.waitForChanges = function(time) {  
  return new Promise(resolve => {  
    this.waiting.push(resolve);  
    setTimeout(() => {  
      if (!this.waiting.includes(resolve)) return;  
      this.waiting = this.waiting.filter(r => r !== resolve);  
      resolve({status: 304});  
    }, time * 1000);  
  });  
};
```

Registrar un cambio con `updated` incrementa la propiedad `versión` y despierta todas las solicitudes en espera.

```
SkillShareServer.prototype.updated = function() {  
  this.version++;  
  let response = this.talkResponse();  
  this.waiting.forEach(resolve => resolve(response));  
  this.waiting = [];  
};
```

Eso concluye el código del servidor. Si creamos una instancia de `SkillShareServer` y la iniciamos en el puerto 8000, el servidor HTTP resultante servirá archivos desde el subdirectorio `public` junto con una interfaz para manejar charlas bajo la URL `/talks`.

```
new SkillShareServer({}).start(8000);
```

EL CLIENTE

La parte del cliente del sitio web de intercambio de habilidades consiste en tres archivos: una pequeña página HTML, una hoja de estilos y un archivo JavaScript.

HTML

Es una convención ampliamente utilizada para servidores web intentar servir un archivo llamado `index.html` cuando se realiza una solicitud directamente a una ruta que corresponde a un directorio. El módulo de servidor de archivos que utilizamos, `serve-static`, soporta esta convención. Cuando se realiza una solicitud a la ruta `/`, el servidor busca el archivo `./public/index.html` (`./public` siendo la raíz que le dimos) y devuelve ese archivo si se encuentra.

Por lo tanto, si queremos que una página aparezca cuando un navegador apunta a nuestro servidor, deberíamos colocarla en `public/index.html`. Este es nuestro archivo de índice:

```
<!doctype html>
<meta charset="utf-8">
<title>Intercambio de habilidades</title>
<link rel="stylesheet" href="skillsharing.css">
```

```
<h1>Intercambio de habilidades</h1>
```

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

Define el título del documento e incluye una hoja de estilos, que define algunos estilos para, entre otras cosas, asegurarse de que haya algo de espacio entre las charlas. Luego agrega un encabezado en la parte superior de la página y carga el script que contiene la aplicación del cliente.

ACCIONES

El estado de la aplicación consiste en la lista de charlas y el nombre del usuario, y lo almacenaremos en un objeto `{charlas, usuario}`. No permitimos que la interfaz de usuario manipule directamente el estado ni envíe solicitudes HTTP. En cambio, puede emitir *acciones* que describen lo que el usuario está intentando hacer.

La función `handleAction` toma una acción de este tipo y la lleva a cabo. Debido a que nuestras actualizaciones de estado son tan simples, los cambios de estado se manejan en la misma función.

```
function handleAction(state, action) {
  if (action.type == "setUser") {
    localStorage.setItem("userName", action.user);
    return {...state, user: action.user};
  } else if (action.type == "setTalks") {
    return {...state, talks: action.talks};
  } else if (action.type == "newTalk") {
    fetchOK(talkURL(action.title), {
      method: "PUT",
      headers: {"Content-Type": "application/json"},
      body: JSON.stringify({
```



```

        presenter: state.user,
        summary: action.summary
    })
  }).catch(reportError);
} else if (action.type == "deleteTalk") {
  fetchOK(talkURL(action.talk), {method: "DELETE"})
    .catch(reportError);
} else if (action.type == "newComment") {
  fetchOK(talkURL(action.talk) + "/comments", {
    method: "POST",
    headers: {"Content-Type": "application/json"},
    body: JSON.stringify({
      author: state.user,
      message: action.message
    })
  }).catch(reportError);
}
return state;
}

```

Almacenaremos el nombre del usuario en `localStorage` para que pueda ser restaurado cuando se cargue la página.

Las acciones que necesitan involucrar al servidor realizan peticiones a la red, utilizando `fetch`, a la interfaz HTTP descrita anteriormente. Utilizamos una función de envoltura, `fetchOK`, que se asegura de que la promesa devuelta sea rechazada cuando el servidor devuelve un código de error.

```

function fetchOK(url, options) {
  return fetch(url, options).then(response => {
    if (response.status < 400) return response;
    else throw new Error(response.statusText);
  });
}

```

Esta función auxiliar se utiliza para construir una URL para una charla con un título dado.

```
function talkURL(title) {  
  return "talks/" + encodeURIComponent(title);  
}
```

Cuando la petición falla, no queremos que nuestra página simplemente se quede ahí, sin hacer nada sin explicación. Así que definimos una función llamada `reportError`, que al menos muestra al usuario un cuadro de diálogo que le informa que algo salió mal.

```
function reportError(error) {  
  alert(String(error));  
}
```

RENDERIZACIÓN DE COMPONENTES

Utilizaremos un enfoque similar al que vimos en [Capítulo 19](#), dividiendo la aplicación en componentes. Pero dado que algunos de los componentes nunca necesitan actualizarse o siempre se redibujan por completo cuando se actualizan, definiremos aquellos no como clases, sino como funciones que devuelven directamente un nodo DOM. Por ejemplo, aquí hay un componente que muestra el campo donde el usuario puede ingresar su nombre:

```
function renderUserField(name, dispatch) {  
  return elt("label", {}, "Tu nombre: ", elt("input", {  
    type: "text",  
    value: name,  
    onchange(event) {  
      dispatch({type: "setUser", user: event.target.value});  
    }  
  }));  
}
```

La función `elt` utilizada para construir elementos DOM es la misma que usamos en [Capítulo 19](#).

Se utiliza una función similar para renderizar charlas, que incluyen una lista de comentarios y un formulario para agregar un nuevo comentario.

```
function renderTalk(talk, dispatch) {
  return elt(
    "section", {className: "talk"},
    elt("h2", null, talk.title, " ", elt("button", {
      type: "button",
      onclick() {
        dispatch({type: "deleteTalk", talk: talk.title});
      }
    }, "Eliminar")),
    elt("div", null, "por ",
      elt("strong", null, talk.presenter)),
    elt("p", null, talk.summary),
    ...talk.comments.map(renderComment),
    elt("form", {
      onsubmit(event) {
        event.preventDefault();
        let form = event.target;
        dispatch({type: "newComment",
          talk: talk.title,
          message: form.elements.comment.value});
        form.reset();
      }
    }, elt("input", {type: "text", name: "comment"}), " ",
      elt("button", {type: "submit"}, "Añadir comentario")));
}
```

El controlador de evento `"submit"` llama a `form.reset` para limpiar el contenido del formulario después de crear una acción `"newComment"`.

Cuando se crean piezas moderadamente complejas del DOM, este estilo de programación comienza a verse bastante desordenado. Para evitar esto, a menudo la gente utiliza un *lenguaje de plantillas*, que permite escribir la interfaz como un archivo HTML con algunos marcadores especiales para indicar dónde van los elementos dinámicos. O utilizan *JSX*, un dialecto de JavaScript no estándar que te permite escribir algo muy parecido a etiquetas HTML en tu programa como si fueran expresiones JavaScript. Ambos enfoques utilizan herramientas adicionales para preprocesar el código antes de que pueda ser ejecutado, lo cual evitaremos en este capítulo.

Los comentarios son simples de renderizar.

```
function renderComment(comment) {
  return elt("p", {className: "comment"},
    elt("strong", null, comment.author),
    ": ", comment.message);
}
```

Finalmente, el formulario que el usuario puede usar para crear una nueva charla se representa de la siguiente manera:

```
function renderTalkForm(dispatch) {
  let title = elt("input", {type: "text"});
  let summary = elt("input", {type: "text"});
  return elt("form", {
    onsubmit(event) {
      event.preventDefault();
      dispatch({type: "newTalk",
        title: title.value,
        summary: summary.value});
      event.target.reset();
    }
  }, elt("h3", null, "Enviar una charla"),
    elt("label", null, "Título: ", title),
    elt("label", null, "Resumen: ", summary),
```

```
    elt("button", {type: "submit"}, "Enviar"));
}
```

SONDEO

Para iniciar la aplicación necesitamos la lista actual de charlas. Dado que la carga inicial está estrechamente relacionada con el proceso de sondeo prolongado, el ETag de la carga debe ser utilizado al sondear, escribiremos una función que siga sondeando al servidor en busca de `/charlas` y llame a una función de devolución de llamada cuando un nuevo conjunto de charlas esté disponible.

```
async function pollTalks(update) {
  let tag = undefined;
  for (;;) {
    let response;
    try {
      response = await fetchOK("/charlas", {
        headers: tag && {"If-None-Match": tag,
          "Prefer": "wait=90"}
      });
    } catch (e) {
      console.log("La solicitud falló: " + e);
      await new Promise(resolve => setTimeout(resolve, 500));
      continue;
    }
    if (response.status == 304) continue;
    tag = response.headers.get("ETag");
    update(await response.json());
  }
}
```

Esta es una función `async` para facilitar el bucle y la espera de la solicitud. Ejecuta un bucle infinito que, en cada iteración, recupera la lista de charlas, ya sea normalmente o, si esta no es la primera

solicitud, con las cabeceras incluidas que la convierten en una solicitud de sondeo prolongado.

Cuando una solicitud falla, la función espera un momento y luego intenta nuevamente. De esta manera, si tu conexión de red se interrumpe por un tiempo y luego vuelve, la aplicación puede recuperarse y continuar actualizándose. La promesa resuelta a través de `setTimeout` es una forma de forzar a la función `async` a esperar.

Cuando el servidor devuelve una respuesta 304, eso significa que una solicitud de intercambio de larga duración expiró, por lo que la función debería comenzar inmediatamente la siguiente solicitud. Si la respuesta es un estado 200 normal, su cuerpo se lee como JSON y se pasa a la devolución de llamada, y el valor del encabezado `ETag` se almacena para la próxima iteración.

LA APLICACIÓN

El siguiente componente une toda la interfaz de usuario:

```
class SkillShareApp {
  constructor(state, dispatch) {
    this.dispatch = dispatch;
    this.talkDOM = elt("div", {className: "talks"});
    this.dom = elt("div", null,
      renderUserField(state.user, dispatch),
      this.talkDOM,
      renderTalkForm(dispatch));
    this.syncState(state);
  }

  syncState(state) {
    if (state.talks !== this.talks) {
      this.talkDOM.textContent = "";
    }
  }
}
```

```

    for (let talk of state.talks) {
      this.talkDOM.appendChild(
        renderTalk(talk, this.dispatch));
    }
    this.talks = state.talks;
  }
}
}

```

Cuando las charlas cambian, este componente las vuelve a dibujar todas. Esto es simple pero también derrochador. Hablaremos sobre eso en los ejercicios.

Podemos iniciar la aplicación de esta manera:

```

function runApp() {
  let user = localStorage.getItem("userName") || "Anon";
  let state, app;
  function dispatch(action) {
    state = handleAction(state, action);
    app.syncState(state);
  }

  pollTalks(talks => {
    if (!app) {
      state = {user, talks};
      app = new SkillShareApp(state, dispatch);
      document.body.appendChild(app.dom);
    } else {
      dispatch({type: "setTalks", talks});
    }
  }).catch(reportError);
}

runApp();

```

Si ejecutas el servidor y abres dos ventanas del navegador para <http://localhost:8000> una al lado de la otra, puedes ver que las

acciones que realizas en una ventana son inmediatamente visibles en la otra.

EJERCICIOS

Los siguientes ejercicios implicarán modificar el sistema definido en este capítulo. Para trabajar en ellos, asegúrate de descargar primero el código (<https://eloquentjavascript.net/code/skillsharing.zip>), tener Node instalado (<https://nodejs.org>), e instalar la dependencia del proyecto con `npm install`.

PERSISTENCIA EN DISCO

El servidor de intercambio de habilidades mantiene sus datos puramente en memoria. Esto significa que cuando se produce un fallo o se reinicia por cualquier motivo, se pierden todas las charlas y comentarios.

Extiende el servidor para que almacene los datos de las charlas en disco y vuelva a cargar automáticamente los datos cuando se reinicie. No te preocupes por la eficiencia, haz lo más simple que funcione.

RESTABLECIMIENTO DEL CAMPO DE COMENTARIOS

La remodelación completa de las charlas funciona bastante bien porque generalmente no se puede distinguir entre un nodo de DOM y su sustitución idéntica. Pero hay excepciones. Si empiezas a escribir algo en el campo de comentarios para una charla en una ventana del navegador y luego, en otra, añades un comentario a esa charla, el campo en la primera ventana se volverá a dibujar, eliminando tanto su contenido como su enfoque.

Cuando varias personas están añadiendo comentarios al mismo tiempo, esto podría resultar molesto. ¿Puedes idear una manera de resolverlo?

EXERCISE HINTS

The hints below might help when you are stuck with one of the exercises in this book. They don't give away the entire solution, but rather try to help you find it yourself.

ESTRUCTURA DEL PROGRAMA

HACIENDO UN TRIÁNGULO CON BUCLES

Puedes comenzar con un programa que imprime los números del 1 al 7, el cual puedes obtener haciendo algunas modificaciones al ejemplo de impresión de números pares dado anteriormente en el capítulo, donde se introdujo el bucle `for`.

Ahora considera la equivalencia entre los números y las cadenas de caracteres `"#"`. Puedes pasar de 1 a 2 sumando 1 (`+= 1`). Puedes pasar de `"#"` a `"##"` agregando un carácter (`+= "#"`). Por lo tanto, tu solución puede seguir de cerca el programa de impresión de números.

FIZZBUZZ

Claramente, recorrer los números es un trabajo de bucle, y seleccionar qué imprimir es una cuestión de ejecución condicional. Recuerda el truco de usar el operador de resto (`%`) para verificar si un número es divisible por otro número (tiene un resto de cero).

En la primera versión, hay tres resultados posibles para cada número, por lo que tendrás que crear una cadena `if/else if/else`.

La segunda versión del programa tiene una solución sencilla y una inteligente. La solución simple es agregar otra “rama” condicional para probar exactamente la condición dada. Para la solución inteligente, construye una cadena que contenga la palabra o palabras a imprimir e imprime esta palabra o el número si no hay palabra, potencialmente haciendo un buen uso del operador `||`.

TABLERO DE AJEDREZ

Para trabajar con dos dimensiones, necesitarás un bucle dentro de otro bucle. Pon llaves alrededor de los cuerpos de ambos bucles para que sea fácil ver dónde empiezan y terminan. Intenta indentar correctamente estos cuerpos. El orden de los bucles debe seguir el orden en el que construimos la cadena (línea por línea, de izquierda a derecha, de arriba abajo). Entonces el bucle exterior maneja las líneas y el bucle interior maneja los caracteres en una línea.

Necesitarás dos variables para hacer un seguimiento de tu progreso. Para saber si debes colocar un espacio o un signo de hash en una posición determinada, podrías verificar si la suma de los dos contadores es par (`% 2`).

Terminar una línea agregando un carácter de salto de línea debe ocurrir después de que se haya construido la línea, así que hazlo después del bucle interno pero dentro del bucle externo.

FUNCIONES

MÍNIMO

Si tienes problemas para colocar llaves y paréntesis en el lugar correcto para obtener una definición de función válida, comienza copiando uno de los ejemplos de este capítulo y modifícalo.

Una función puede contener múltiples declaraciones `return`.

RECURSIÓN

Es probable que tu función se parezca en cierta medida a la función interna `find` en el ejemplo recursivo `findSolution` [ejemplo](#) de este capítulo, con una cadena `if/else if/else` que prueba cuál de los tres casos aplica. El `else` final, correspondiente al tercer caso, realiza la llamada recursiva. Cada una de las ramas debe contener una declaración `return` o de alguna otra manera asegurarse de que se devuelva un valor específico.

Cuando se le da un número negativo, la función se llamará recursivamente una y otra vez, pasándose a sí misma un número cada vez más negativo, alejándose así más y más de devolver un resultado. Eventualmente se quedará sin espacio en la pila y se abortará.

CONTANDO FRIJOLES

Tu función necesita un bucle que mire cada carácter en la cadena. Puede ejecutar un índice desde cero hasta uno menos que su longitud (`< string.length`). Si el carácter en la posición actual es el mismo que el que la función está buscando, agrega 1 a una

variable de contador. Una vez que el bucle ha terminado, el contador puede ser devuelto.

Ten cuidado de que todas las vinculaciones utilizadas en la función sean *locales* a la función, declarándolas correctamente con la palabra clave `let` o `const`.

ESTRUCTURAS DE DATOS: OBJETOS Y ARRAYS

LA SUMA DE UN RANGO

La construcción de un array se hace más fácilmente inicializando primero un enlace a `[]` (un array vacío nuevo) y llamando repetidamente a su método `push` para agregar un valor. No olvides devolver el array al final de la función.

Dado que el límite final es inclusivo, necesitarás usar el operador `<=` en lugar de `<` para verificar el final de tu bucle.

El parámetro de paso puede ser un parámetro opcional que por defecto (usando el operador `=`) es 1.

Hacer que `range` comprenda valores negativos de paso probablemente sea mejor haciendo escribiendo dos bucles separados: uno para contar hacia arriba y otro para contar hacia abajo, porque la comparación que verifica si el bucle ha terminado necesita ser `>=` en lugar de `<=` al contar hacia abajo.

También puede valer la pena usar un paso predeterminado diferente, es decir, `-1`, cuando el final del rango es menor que el principio. De esa manera, `range(5, 2)` devuelve algo significativo, en lugar de

quedarse atascado en un bucle infinito. Es posible hacer referencia a parámetros anteriores en el valor predeterminado de un parámetro.

REVERSIÓN DE UN ARRAY

Hay dos formas obvias de implementar `reverseArray`. La primera es simplemente recorrer el array de entrada de principio a fin y usar el método `unshift` en el nuevo array para insertar cada elemento en su inicio. La segunda es recorrer el array de entrada hacia atrás y utilizar el método `push`. Iterar sobre un array hacia atrás requiere una especificación de bucle (algo incómoda), como `(let i = array.length - 1; i >= 0; i--)`.

Invertir el array en su lugar es más difícil. Debes tener cuidado de no sobrescribir elementos que necesitarás más adelante. Utilizar `reverseArray` o copiar todo el array de otra manera (usar `array.slice()` es una buena forma de copiar un array) funciona pero es hacer trampa.

El truco consiste en *intercambiar* el primer y último elementos, luego el segundo y el penúltimo, y así sucesivamente. Puedes hacer esto recorriendo la mitad de la longitud del array (utiliza `Math.floor` para redondear hacia abajo, no necesitas tocar el elemento central en un array con un número impar de elementos) e intercambiando el elemento en la posición `i` con el que está en la posición `array.length - 1 - i`. Puedes utilizar una asignación local para retener brevemente uno de los elementos, sobrescribirlo con su imagen reflejada, y luego colocar el valor de la asignación local en el lugar donde solía estar la imagen reflejada.

LISTA

Construir una lista es más fácil cuando se hace de atrás hacia adelante. Por lo tanto, `arrayToList` podría iterar sobre el array en reversa (ver ejercicio anterior) y, para cada elemento, agregar un objeto a la lista. Puedes usar un enlace local para mantener la parte de la lista que se ha construido hasta el momento y usar una asignación como `lista = {value: X, rest: lista}` para añadir un elemento.

Para recorrer una lista (en `listToArray` y `nth`), se puede utilizar una especificación de bucle `for` de esta forma:

```
for (let nodo = lista; nodo; nodo = nodo.rest) {}
```

¿Puedes ver cómo funciona esto? En cada iteración del bucle, `nodo` apunta a la sublista actual, y el cuerpo puede leer su propiedad `value` para obtener el elemento actual. Al final de una iteración, `nodo` pasa a la siguiente sublista. Cuando eso es nulo, hemos llegado al final de la lista y el bucle ha terminado.

La versión recursiva de `nth` mirará de manera similar una parte cada vez más pequeña de la “cola” de la lista y al mismo tiempo contará hacia abajo el índice hasta llegar a cero, momento en el que puede devolver la propiedad `value` del nodo que está observando. Para obtener el elemento cero de una lista, simplemente tomas la propiedad `value` de su nodo principal. Para obtener el elemento $N + 1$, tomas el elemento N -ésimo de la lista que se encuentra en la propiedad `rest` de esta lista.

COMPARACIÓN PROFUNDA

La prueba para determinar si estás tratando con un objeto real se verá algo así: `typeof x == "object" && x != null`. Ten cuidado de comparar propiedades solo cuando *ambos* argumentos sean objetos. En todos los demás casos, simplemente puedes devolver inmediatamente el resultado de aplicar `===`.

Utiliza `Object.keys` para recorrer las propiedades. Necesitas comprobar si ambos objetos tienen el mismo conjunto de nombres de propiedades y si esas propiedades tienen valores idénticos. Una forma de hacerlo es asegurarse de que ambos objetos tengan el mismo número de propiedades (las longitudes de las listas de propiedades son iguales). Y luego, al recorrer las propiedades de uno de los objetos para compararlas, asegúrate siempre primero de que el otro realmente tenga una propiedad con ese nombre. Si tienen el mismo número de propiedades y todas las propiedades en uno también existen en el otro, tienen el mismo conjunto de nombres de propiedades.

Devolver el valor correcto de la función se hace mejor devolviendo inmediatamente `false` cuando se encuentra una diferencia y devolviendo `true` al final de la función.

FUNCIONES DE ORDEN SUPERIOR

EVERYTHING

Como el operador `&&`, el método `every` puede dejar de evaluar más elementos tan pronto como encuentre uno que no coincida. Por lo tanto, la versión basada en bucle puede salir del bucle—con `break` o ``return``—tan pronto como encuentre un elemento para el que la función de predicado devuelva `false`. Si el bucle se ejecuta hasta el

final sin encontrar dicho elemento, sabemos que todos los elementos coincidieron y deberíamos devolver `true`.

Para construir `every` sobre `some`, podemos aplicar *leyes de De Morgan*, que establecen que `a && b` es igual a `!(!a || !b)`. Esto se puede generalizar a arrays, donde todos los elementos en el array coinciden si no hay ningún elemento en el array que no coincida.

DIRECCIÓN DE ESCRITURA DOMINANTE

Tu solución podría parecerse mucho a la primera mitad del ejemplo de `textScripts`. De nuevo, debes contar caracteres según un criterio basado en `characterScript` y luego filtrar la parte del resultado que se refiere a caracteres no interesantes (sin script).

Encontrar la dirección con el recuento de caracteres más alto se puede hacer con `reduce`. Si no está claro cómo hacerlo, consulta el ejemplo anterior en el capítulo, donde se usó `reduce` para encontrar el script con más caracteres.

LA VIDA SECRETA DE LOS OBJETOS

UN TIPO DE VECTOR

Mira de nuevo el ejemplo de la clase `Rabbit` si no estás seguro de cómo se ven las declaraciones de `class`.

Agregar una propiedad getter al constructor se puede hacer poniendo la palabra `get` antes del nombre del método. Para calcular la distancia desde $(0, 0)$ hasta (x, y) , puedes usar el teorema de Pitágoras, que dice que el cuadrado de la distancia que estamos buscando es igual al cuadrado de la coordenada x más el cuadrado de

la coordenada y . Por lo tanto, $\sqrt{x^2 + y^2}$ es el número que buscas. `Math.sqrt` es la forma de calcular una raíz cuadrada en JavaScript y `x ** 2` se puede usar para elevar al cuadrado un número.

GRUPOS

La forma más sencilla de hacer esto es almacenar un array de miembros del grupo en una propiedad de instancia. Los métodos `includes` o `indexOf` se pueden usar para verificar si un valor dado está en el array.

El constructor de tu clase puede establecer la colección de miembros en un array vacío. Cuando se llama a `add`, debe verificar si el valor dado está en el array o agregarlo, por ejemplo con `push`, de lo contrario.

Eliminar un elemento de un array, en `delete`, es menos directo, pero puedes usar `filter` para crear un nuevo array sin el valor. No olvides sobrescribir la propiedad que contiene los miembros con la nueva versión filtrada del array.

El método `from` puede usar un bucle `for/of` para obtener los valores del objeto iterable y llamar a `add` para colocarlos en un grupo recién creado.

GRUPOS ITERABLES

Probablemente valga la pena definir una nueva clase `GroupIterator`. Las instancias del iterador deberían tener una propiedad que rastree la posición actual en el grupo. Cada vez que se

llama a `next`, verifica si ha terminado y, si no, avanza más allá del valor actual y lo devuelve.

La clase `Group` en sí misma obtiene un método nombrado `Symbol.iterator` que, al ser llamado, devuelve una nueva instancia de la clase iteradora para ese grupo.

PROYECTO: UN ROBOT

MEDICIÓN DE UN ROBOT

Tendrás que escribir una variante de la función `runRobot` que, en lugar de registrar los eventos en la consola, devuelva el número de pasos que el robot tomó para completar la tarea.

Tu función de medición puede, entonces, en un bucle, generar nuevos estados y contar los pasos que toma cada uno de los robots. Cuando haya generado suficientes mediciones, puede usar `console.log` para mostrar el promedio de cada robot, que es el número total de pasos tomados dividido por el número de mediciones.

EFICIENCIA DEL ROBOT

La principal limitación de `goalOrientedRobot` es que solo considera un paquete a la vez. A menudo caminará de un lado a otro del pueblo porque el paquete en el que está centrando su atención sucede que está en el otro lado del mapa, incluso si hay otros mucho más cerca.

Una posible solución sería calcular rutas para todos paquetes y luego tomar la más corta. Se pueden obtener resultados aún mejores, si

hay múltiples rutas más cortas, al preferir aquellas que van a recoger un paquete en lugar de entregarlo.

GRUPO PERSISTENTE

La forma más conveniente de representar el conjunto de valores miembro sigue siendo como un array, ya que los arrays son fáciles de copiar.

Cuando se añade un valor al grupo, puedes crear un nuevo grupo con una copia del array original que tenga el valor añadido (por ejemplo, usando `concat`). Cuando se elimina un valor, puedes filtrarlo del array.

El constructor de la clase puede tomar dicho array como argumento y almacenarlo como propiedad única de la instancia. Este array nunca se actualiza.

Para añadir la propiedad `empty` al constructor, puedes declararla como una propiedad estática.

Solo necesitas una instancia `empty` porque todos los grupos vacíos son iguales y las instancias de la clase no cambian. Puedes crear muchos grupos diferentes a partir de ese único grupo vacío sin afectarlo.

BUGS Y ERRORES

REINTENTAR

La llamada a `primitiveMultiply` definitivamente debería ocurrir en un bloque `try`. El bloque `catch` correspondiente debería

relanzar la excepción cuando no sea una instancia de `MultiplierUnitFailure` y asegurarse de que la llamada se reintente cuando lo sea.

Para hacer el reintentamiento, puedes usar un bucle que se detenga solo cuando una llamada tiene éxito, como en el ejemplo de [look](#) anterior en este capítulo, o usar la recursión y esperar que no tengas una cadena tan larga de fallos que colapse la pila (lo cual es bastante improbable).

LA CAJA CERRADA CON LLAVE

En este ejercicio, es posible que desees usar `try` y `finally` juntos. Tu función debería desbloquear la caja y luego llamar a la función de argumento desde dentro de un bloque `try`. El bloque `finally` después de él debería volver a bloquear la caja.

Para asegurarte de que no bloquee la caja cuando no estaba bloqueada, verifica su bloqueo al comienzo de la función y desbloquéala y bloquéala solo cuando comenzó bloqueada.

EXPRESIONES REGULARES

ESTILO DE COMILLAS

La solución más obvia es reemplazar solo las comillas que tienen un carácter que no sea una letra en al menos un lado, algo como `/\P{L}' | '\P{L}/`. Pero también debes tener en cuenta el inicio y el final de la línea.

Además, debes asegurarte de que la sustitución también incluya los caracteres que coincidieron con el patrón `\P{L}` para que no se

eliminen. Esto se puede hacer envolviéndolos entre paréntesis e incluyendo sus grupos en la cadena de reemplazo ($\$1$, $\$2$). Los grupos que no se emparejen se reemplazarán por nada.

NÚMEROS NUEVAMENTE

Primero, no olvides la barra invertida delante del punto.

Para hacer coincidir el signo opcional delante del número, así como delante del exponente, se puede hacer con $[+\backslash-]?$ o $(\backslash+|-|)$ (más, menos, o nada).

La parte más complicada del ejercicio es el problema de hacer coincidir tanto $"5."$ como $".5"$ sin hacer coincidir también $"."$. Para esto, una buena solución es usar el operador $|$ para separar los dos casos: uno o más dígitos seguidos opcionalmente por un punto y cero o más dígitos o un punto seguido por uno o más dígitos.

Finalmente, para hacer que el caso de la e sea insensible a mayúsculas y minúsculas, añade una opción i a la expresión regular o usa $[eE]$.

MÓDULOS

UN ROBOT MODULAR

Esto es lo que habría hecho (pero de nuevo, no hay una única forma *correcta* de diseñar un módulo dado):

El código utilizado para construir el gráfico de carreteras se encuentra en el módulo `graph`. Como preferiría usar `dijkstra.js` de NPM en lugar de nuestro propio código de búsqueda de caminos,

haremos que este construya el tipo de datos de gráfico que espera `dijkstra.js`. Este módulo exporta una única función, `buildGraph`. Haría que `buildGraph` aceptara un arreglo de arreglos de dos elementos, en lugar de cuerdas que contienen guiones, para hacer que el módulo dependa menos del formato de entrada.

El módulo `roads` contiene los datos crudos de las carreteras (el arreglo `roads`) y el enlace `roadGraph`. Este módulo depende de `./graph.js` y exporta el grafo de carreteras.

La clase `VillageState` se encuentra en el módulo `state`. Depende del módulo `./roads` porque necesita poder verificar que una carretera dada exista. También necesita `randomPick`. Dado que es una función de tres líneas, podríamos simplemente ponerla en el módulo `state` como una función auxiliar interna. Pero `randomRobot` también la necesita. Entonces tendríamos que duplicarla o ponerla en su propio módulo. Dado que esta función existe en NPM en el paquete `random-item`, una solución razonable es hacer que ambos módulos dependan de eso. También podemos agregar la función `runRobot` a este módulo, ya que es pequeña y está relacionada con la gestión del estado. El módulo exporta tanto la clase `VillageState` como la función `runRobot`.

Finalmente, los robots, junto con los valores en los que dependen, como `mailRoute`, podrían ir en un módulo `example-robots`, que depende de `./roads` y exporta las funciones del robot. Para que `goalOrientedRobot` pueda realizar la búsqueda de rutas, este módulo también depende de `dijkstra.js`. Al externalizar cierto trabajo a módulos NPM, el código se volvió un poco más pequeño.

Cada módulo individual hace algo bastante simple y se puede leer por sí solo. Dividir el código en módulos a menudo sugiere mejoras adicionales en el diseño del programa. En este caso, parece un poco extraño que el `VillageState` y los robots dependan de un gráfico de caminos específico. Podría ser una mejor idea hacer que el gráfico sea un argumento del constructor de estado y hacer que los robots lo lean desde el objeto de estado, esto reduce las dependencias (lo cual siempre es bueno) y hace posible ejecutar simulaciones en mapas diferentes (lo cual es aun mejor).

¿Es una buena idea utilizar módulos de NPM para cosas que podríamos haber escrito nosotros mismos? En principio, sí, para cosas no triviales como la función de búsqueda de caminos es probable que cometas errores y pierdas tiempo escribiéndolas tú mismo. Para funciones pequeñas como `random-item`, escribirlas por ti mismo es bastante fácil. Pero añadirlas donde las necesitas tiende a saturar tus módulos.

Sin embargo, tampoco debes subestimar el trabajo involucrado en *encontrar* un paquete de NPM apropiado. Y aunque encuentres uno, podría no funcionar bien o le podrían faltar alguna característica que necesitas. Además, depender de paquetes de NPM significa que debes asegurarte de que estén instalados, debes distribuirlos con tu programa y es posible que debas actualizarlos periódicamente.

Así que de nuevo, esto es un compromiso, y puedes decidir de cualquier manera dependiendo de cuánto te ayude realmente un paquete dado.

MÓDULO DE CAMINOS

Dado que este es un módulo ES, debes usar `import` para acceder al módulo de gráfico. Esto se describió como exportando una función de `buildGraph`, la cual puedes seleccionar de su objeto de interfaz con una declaración de desestructuración `const`.

Para exportar `roadGraph`, colocas la palabra clave `export` antes de su definición. Debido a que `buildGraph` toma una estructura de datos que no coincide exactamente con `roads`, la división de las cadenas de carretera debe ocurrir en tu módulo.

DEPENDENCIAS CIRCULARES

El truco es que `require` añade el objeto de interfaz de un módulo a su caché *antes* de comenzar a cargar el módulo. De esta manera, si se hace alguna llamada a `require` mientras se está ejecutando tratando de cargarlo, ya se conoce, y se devolverá la interfaz actual, en lugar de comenzar a cargar el módulo nuevamente (lo que eventualmente desbordaría la pila).

PROGRAMACIÓN ASÍNCRONA

MOMENTOS DE TRANQUILIDAD

Necesitarás convertir el contenido de estos archivos en un array. La forma más fácil de hacerlo es utilizando el método `split` en la cadena producida por `textFile`. Ten en cuenta que para los archivos de registro, eso seguirá dándote un array de cadenas, que debes convertir a números antes de pasarlos a `new Date`.

Resumir todos los puntos temporales en una tabla de horas se puede hacer creando una tabla (array) que contenga un número para cada

hora del día. Luego puedes recorrer todos los marca de tiempos (sobre los archivos de registro y los números en cada archivo de registro) y, para cada uno, si sucedió en el día correcto, toma la hora en que ocurrió y suma uno al número correspondiente en la tabla.

Asegúrate de usar `await` en el resultado de las funciones asíncronas antes de hacer cualquier cosa con él, o terminarás con una `Promise` donde esperabas un string.

hinting}}

PROMESAS REALES

Reescribe la función del ejercicio anterior sin `async/await`, utilizando métodos simples de `Promise`.

En este estilo, usar `Promise.all` será más conveniente que intentar modelar un bucle sobre los archivos de registro. En la función `async`, simplemente usar `await` en un bucle es más simple. Si leer un archivo toma un tiempo, ¿cuál de estos dos enfoques tomará menos tiempo para ejecutarse?

Si uno de los archivos listados en la lista de archivos tiene un error tipográfico, y falla al leerlo, ¿cómo termina ese fallo en el objeto `Promise` que retorna tu función?