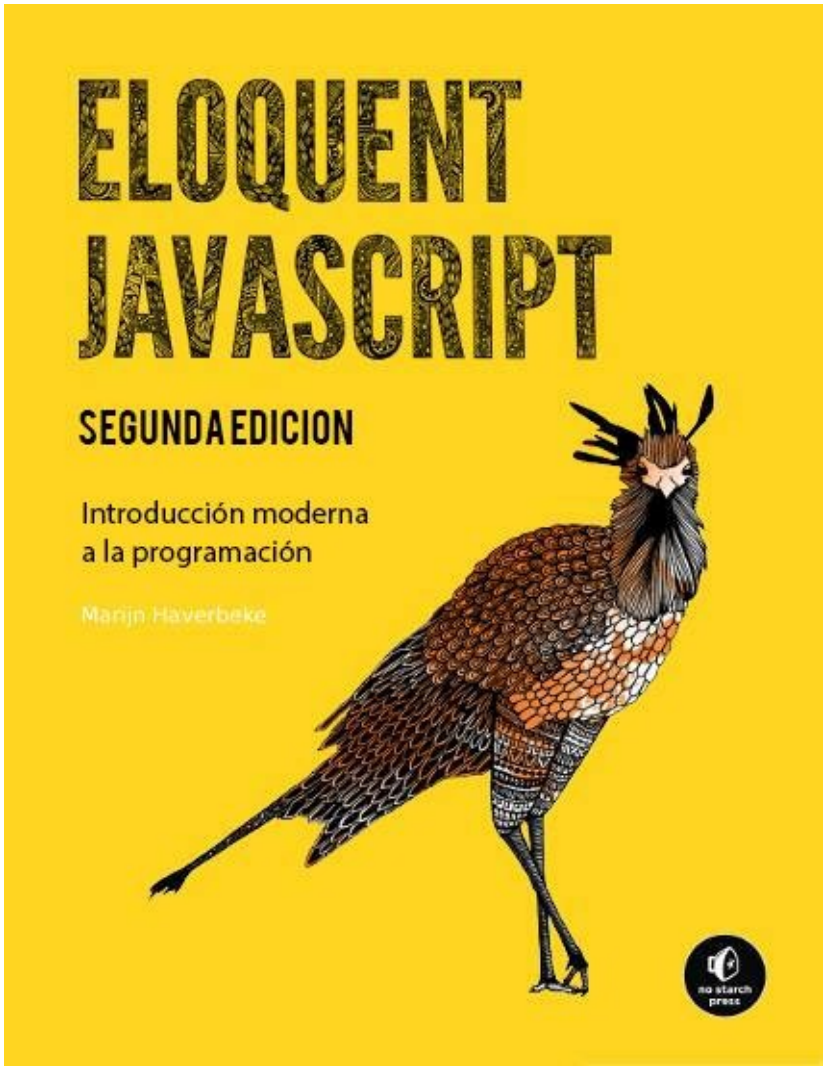
Table of Contents

Eloquent Javascript en Español



Introducción

Este libro trata de como hacer que las computadoras hagan lo que tú quieres que hagan. Las computadoras son tan comunes como los desatornilladores hoy en día, pero tienen mucha más complejidad oculta y, por lo tanto, son más difíciles de operar y entender. Para muchos siguen siendo cosas extrañas, un poco amenazadoras.

Hemos encontrado dos formas efectivas de cerrar la brecha ente nosotros, suaves organismos biológicos con talento para el razonamiento social y espacial, y las computadoras, manipuladoras sin sentimientos de datos sin sentido. La primera es usar nuestros sentidos del mundo físico y construir interfaces que imitan ese mundo y nos permiten manipular figuras en una pantalla con nuestros dedos. Esto funciona muy bien para interacciones casuales con la máquina.

Pero aún no hemos encontrado una buena forma de usar la interfaz gráfica para comunicar a la computadora cosas que el diseñador de interfaces no anticipó. Para interfaces abiertas, como indicarle a la computadora que ejecute tareas arbitrarias, hemos tenido más suerte con otra estrategia, que hace uso de nuestro talento para el lenguaje: enseñarle a la computadora uno.

Los lenguajes humanos permiten que palabras y frases se combinen en muchas diferentes formas, lo cuál nos permite decir una amplia variedad de cosas. Los lenguajes computacionales, aunque son gramaticalmente menos flexibles, siguen un principio similar.

La computación casual se ha extendido mucho en los últimos 20 años, y las interfaces basadas en el lenguaje, que alguna vez fueron la forma predeterminada en la que las personas interactuaban con las computadoras, han sido reemplazadas en gran medida por interfaces gráficas. Pero todavía están ahí, si sabes donde buscar. Uno de estos lenguajes, JavaScript, está presente en casi todos los navegadores web existentes y por lo tanto, disponible en prácticamente todos los dispositivos de consumo.

Este libro trata de hacer que te familiarices lo suficiente con este lenguaje para que puedas hacer que la computadora haga lo que tú quieras.

En la Programación

> No ilumino a aquellos que no están deseosos de aprender, tampoco despierto a quienes no están ansiosos de darse una explicación a sí mismos. Si les he presentado una esquina del cuadro y ellos no vienen con las otras tres, no debería recorrer otra vez los puntos." Confucio

A parte de explicar JavaScript, te introduciré en los principios básicos de la programación. Programar, resulta difícil. Las reglas fundamentales típicamente son simples y claras. Pero los programas contruidos sobre esas reglas tienden a volverse lo suficientemente complejos para introducir sus propias reglas y más complejidad. En cierta forma, estás construyendo tu propio laberinto y podrías perderte en él.

Habrá ocasiones en las que al leer este libro te sentirás terriblemente frustrado. Si eres nuevo programando, tendrás mucho material nuevo para digerir. Mucho de este material después será *combinado* en formas que requerirán que hagas conexiones adicionales.

Es tu responsabilidad realizar el esfuerzo necesario. Cuando se te haga difícil seguir este libro, no concluyas rápidamente nada acerca de tus capacidades. Tu eres bueno—sólo necesitas mantener el paso. Toma un respiro, vuelve a leer el material, y *siempre* asegúrate de leer y entender los programas de ejemplo y los ejercicios. Aprender es un trabajo duro, pero todo lo que aprendas ahora es tuyo y hará el aprendizaje cada vez más fácil.

> El programador de computadoras es el creador de universos de los cuales él sólo es responsable. Universos de complejidad virtualmente ilimitada pueden ser creados en la forma de programas de computadora."
>
> Joseph Weizenbaum, Computer Power and Human Reason

Un programa es muchas cosas. Es una pieza de texto escrita por un programador, es la fuerza que dirije a la computadora para hacer lo que hace, son datos en la memoria de la computadora, y aún así controla las acciones realizadas en esta misma memoria. Las analogías que tratan de comparar a las computadoras con objetos que conocemos tienden a quedarse cortas. Una que se ajusta superficialmente es la de máquina, un montón de piezas separadas que se relacionan, y para hacerlo funcionar, tenemos que considerar las formas en que esas piezas se interconectan y contribuyen al funcionamiento del todo.

Una computadora es una máquina que actua como anfitriona de estas máquinas inmateriales. Las computadoras por sí mismas sólo pueden hacer cosas estúpidamente simples. La razón por la que son tan poderosas es que hacen esas cosas a una velocidad

increíblemente rápida. Un programa puede combinar ingeniosamente un número enorme de esas acciones simples para lograr cosas muy complicadas.

Para algunos de nosotros, escribir programas de computadoras es un juego fascinante. Un programa es un construcción del pensamiento. No tiene costo construirlo, no pesa nada, y crece fácilmente bajo nuestras manos tecleando.

Pero si no tenemos cuidado, el tamaño y la complejidad de un programa se saldrán de control, confundiendo incluso a la persona que lo creó. Mantener los programas bajo control es el principal problema de la programación. Cuando funcionan, es hermoso. El arte de programar es la habilidad de controlar la complejidad. Un gran programa está dominado, hecho simple en su complejidad.

Muchos programadores creen que esta complejidad es mejor controlada usando sólo un pequeño conjunto de técnicas bien entendidas en sus programas. Estos han compuesto reglas estrictas (“mejores prácticas”) que prescriben la forma que los programas deberían tener, y los más celosos de ellos considerarán a aquellos que se salen de esta pequeña zona segura como *malos* programadores.

¡Qué hostilidad hacia la riqueza de la programación, la de tratar de reducirla a algo simple y predecible, tratar de hacer tabú a todos los programas extraños y bellos! El panorama de todas las técnicas de programación es enorme, fascinante en su diversidad, y permanece inexplorado en gran parte. Ciertamente es peligroso, ira seduciendo al programador novato con todo tipo de confusiones, pero eso sólo significa que debes de andar con cuidado y estar alerta. Conforme vayas aprendiendo, siempre habrá nuevos retos y nuevo territorio por explorar. Los programadores que se nieguen a explorar dejarán de progresar, olvidarán su alegría, y se aburrirán con su trabajo.

Por qué el lenguaje importa

En el principio, cuando nació la computación, no había lenguajes de programación. Los progrmas 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
```

Eso es un programa para sumar los números del 1 al 10 e imprimir el resultado: `1 + 2 + ... + 10 = 55`. Podría correr en una simple, hipotética máquina. Para programar las primeras computadoras, era necesario configurar grandes conjuntos de interruptores en la posición correcta o perforar tiras de tarjetas e introducirlas en la computadora. Probablemente te puedes imaginar cuan tedioso y propenso al error era este procedimiento. Incluso escribir programas simples requería gran inteligencia y disciplina. Los programas complejos eran casi inconcebibles.

Claro, introducir manualmente estos oscuros patrones de bits (los unos y ceros) dieron al programador un profundo sentimiento de ser un poderoso mago. Y eso ha valido algo en términos de satisfacción en el trabajo.

Cada línea del programa anterior contiene una instrucción. Podría ser escrita en español como sigue:

---

1. Guarda el número 0 en la dirección de memoria 0.
2. Guarda el número 1 en la dirección de memoria 1.
3. Guarda el valor de la dirección de memoria 1 en la dirección 2.
4. Resta 11 del valor en la dirección de memoria 2.
5. Si el valor en la dirección de memoria 2 es el número 0, continúa con la instrucción 9.
6. Suma el valor de la dirección de memoria 1 al valor de la dirección de memoria 0.
7. Suma 1 al valor de la dirección de memoria 1.
8. Continúa con la instrucción 3.
9. Devuelve el valor de la dirección de memoria 0.

Aunque eso es más legible que una sopa de bits, sigue sin ser agradable. Podría ayudar usar nombres en los números para las instrucciones y direcciones de memoria.

```
Pon “total” igual a 0.
Pon “conteo” igual a 1.
[bucle]
  Pon “comparación” igual a “conteo”.
  Resta 11 de “comparación”.
  Si “comparación” es cero, continúa en [final].
  Suma “conteo” a “total”.
  Suma 1 a “conteo”.
  Continúa en [bucle].
[final]
Devuelve “total”.
```

¿Puedes entender cómo funciona el programa en este punto? Las primeras dos líneas ponen en dos locaciones de memoria sus valores iniciales: `total` será usado para construir el resultado del cálculo y `conteo` mantendrá el registro del número en el que estamos trabajando en este momento. Las líneas que usan `comparación` son probablemente las más raras. El programa quiere ver si `conteo` es igual a 11 para saber si puede terminar. A causa de que nuestra máquina hipotética es más bien primitiva, solo puede probar si un número es cero y tomar una decisión (o salto) basada en eso. Así que usa la dirección de memoria etiquetada como `comparación` para calcular el valor de `conteo - 11` y toma una decisión basada en ese valor. Las próximas dos líneas suman el valor de `conteo` al resultado e incrementan `conteo` en 1 cada vez que el programa ha decido que `conteo` no es todavía 11.

Este es el mismo programa en JavaScript:

```
var total = 0;
var conteo = 1;
while (conteo <= 10) {
  total += conteo;
  conteo += 1;
}
console.log(total);
// → 55
```

Ejemplo Codepen

Esta versión nos da unas cuantas mejoras más. Y lo más importante es que ya no hay necesidad de especificar la forma en que queremos que nuestro programa salte de atrás para adelante. El constructor del lenguaje `while` se encarga de eso. Continúa ejecutándo el bloque (dentro de las llaves) debajo de él mientras la condición que se le dio se mantenga. Esa condición es `conteo <= 10`, lo que significa `"conteo"` es menor o igual que 10. Ya no tenemos que crear un valor temporal y compararlo con 0, lo cuál era un detalle sin interés para nosotros. Parte del poder de los lenguajes de programación es que estos se encargan de los detalles que no nos interesan.

Al final del programa, después de que la construcción `while` ha terminado, la operación `console.log` es aplicada al resultado para imprimirlo como resultado.

Finalmente, así es como el programa luciría si sucediera que tenemos las convenientes operaciones `range` y `sum` disponibles, una crea una colección de números dentro de un rango y la otra calcula la suma de una colección de números, respectivamente:

```
console.log(sum(range(1, 10)));
// → 55
```

La moraleja de esta historia es que el mismo programa puede ser expresado en formas largas, cortas, legibles e ilegibles. La primera versión del programa era extremadamente difícil de entender, mientras que la última está casi en lenguaje ingles: `log` (registra) la `sum` (suma) del `rango` de números del 1 al 10. Veremos en capítulos posteriores como construir operaciones como `sum` y `range`.)

Un buen lenguaje de programación ayuda al programador al permitirle hablar acerca de las acciones que la computadora tiene que realizar en un nivel más alto. Ayuda a omitir detalles que no nos interesan, provee convenientes bloques de construcción (tales como `while` y `console.log`), y te permite definir tus propios bloques (como `sum` y `range`), y hace fácil componer esos bloques.

¿Qué es JavaScript?

JavaScript fue introducido en 1995 como una forma de añadir programas a las páginas web en el navegador Netscape Navigator. Desde entonces el lenguaje ha sido adoptado por la mayoría de los navegadores más importantes. Ha hecho posibles las aplicaciones web modernas, aplicaciones con las que puedes interactuar directamente, sin hacer recarga de la página para cada acción. Pero también es usado en sitios web más tradicionales para añadirles distintas formas de interactividad y hacerlos más ingeniosos.

Es importante entender que JavaScript no tiene casi nada que ver con el lenguaje de programación llamado Java. El nombre tan parecido fue inspirado por razones de marketing más que de buen juicio. Cuando JavaScript estaba empezando, el lenguaje Java estaba siendo promovido fuertemente y ganando popularidad. Alguien pensó que sería buena idea colgarse de su éxito. Ahora ya nos quedamos con el nombre.

Después de su adopción fuera de Netscape, un documento de estándar fue escrito para describir la forma en que JavaScript debería de trabajar, para asegurarse de que distintos programas que argumentaban soportar JavaScript hablaran realmente del mismo lenguaje. Este documento es llamado el estándar ECMAScript, en honor a la Ecma International Organization, que realizó la estandarización. En la práctica, los términos ECMAScript y JavaScript pueden ser usados indistintamente, son dos nombres para el mismo lenguaje.

Existen aquellos que dirán cosas *terribles* acerca de JavaScript. Muchas de esas cosas son ciertas. La primera vez que tuve que programar algo en JavaScript, rápidamente llegué a despreciarlo. Aceptaba cualquier cosa que yo escribiera pero la interpretaba en una forma completamente distinta a lo que yo quería decir. Esto tenía mucho que ver con el hecho de que yo no tenía idea de lo que estaba haciendo, por supuesto, pero aquí existe un problema real: JavaScript es extremadamente liberal en lo que permite. La idea detrás de este diseño

es que haría la programación en JavaScript más fácil para los principiantes. En realidad, la mayor parte de las veces eso hace más difícil encontrar los errores en tus programas debido a que el sistema no te los señalará.

Esta flexibilidad tiene sus ventajas también. Permite muchas técnicas que son imposibles en otros lenguajes más rígidos, y como verás, puede ser usada para superar algunas de las fallas de JavaScript (por ejemplo en el Capítulo 10). Después de aprender el lenguaje propiamente y de trabajar con él por un tiempo, JavaScript realmente me ha *gustado*.

Han existido varias versiones de JavaScript. ECMAScript versión 3 fue la versión ampliamente soportada en la época de ascención a la dominación de JavaScript, más o menos entre 2000 y 2010. Durante este tiempo, el trabajo fue dirigido a una ambiciosa versión 4, que planeaba varias mejoras radicales y extensiones al lenguaje. Cambiar un lenguaje vivo, ampliamente usado en una forma tan radical resultó ser políticamente difícil, y el trabajo en la versión 4 fue abandonado en 2008, llevando así a la salida de la mucho menos ambiciosa versión 5 en 2009. Nosotros estamos en el punto en el que todos los navegadores mayores soportan la versión 5, que es la versión en la que este libro se enfocará. La versión 6 está en proceso de ser terminada, y algunos navegadores están empezando a soportar nuevas características de esta versión.

Los navegadores web no son las únicas plataformas en las que JavaScript es usado. Algunas Bases de Datos, como MongoDB y CouchDB, usan JavaScript como su lenguaje de manejo y consulta. Varias plataformas para programación de computadoras de escritorio y servidores, más notablemente el proyecto Node.js (el tema del Capítulo 20), están haciendo disponible un entorno poderoso para la programación en JavaScript fuera del navegador.

Código, y qué hacer con él

El código es el texto del que están compuestos los programas. La mayor parte de los capítulos de este libro contienen un montón de él. En mi experiencia, leer y escribir código son partes indispensables de aprender a programar, así que trata de no sólo darles una mirada rápida a los ejemplos. Léelos aténtamente y entiéndelos. Esto podría ser lento y confuso al principio, pero prometo que rápidamente los entenderas. Lo mismo es aplicable a los ejercicios. No asumas que los entiendes hasta que realemente hayas escrito una solución que funcione.

Te recomiendo probar tus soluciones a los ejercicios en un intérprete de JavaScript real. De esa forma obtendrás retroalimentación inmediata acerca de si lo que estás haciendo funciona, y, espero, seas tentado a experimentar e ir más allás de los ejercicios.

La manera más fácil de correr el código del libro, y de experimentar con él, es en la versión web en http://eloquentjavascript.net/. Ahí podrás hacer click en un ejemplo para editarlo y correrlo y para ver la salida que produce. Para trabajar en lo ejercicios, dirígete a http://eloquentjavascript.net/, que prove código para empezar con cada ejercicio y te permite ver las soluciones.

Si quieres correr los programas de este libro fuera del ambiente que se provee, hay que prestarle atención a ciertas cosas. Muchos ejemplos deberían trabajar por sí mismos. Pero el código de los capítulos más avanzados está escrito para un entorno específico (el navegador o Node.js) y sólo puede correr ahí. Además, muchos capítulos definen programas más grandes, y las partes del código que aparecen en él dependen de entre ellas o de archivos externos. El http://eloquentjavascript.net/code en el sitio tiene links para descargar los archivos Zip que contienen todos los scripts y datos necesarios para hacer funcionar el código de cualquier capítulo.

Vista general del libro

Este libro está compuesto por tres partes. Los primeros 11 capítulos hablan de JavaScript en sí mismo. Los siguientes ocho son acerca de los navegadores web y la forma en que JavaScript es usado para programarlos. Finalmente, los últimos dos capítulos están dedicados a ((Node.js)), otro entorno para programar en JavaScript.

A lo largo del libro hay cinco *capítulos de proyecto*, que describen programas de ejemplo más grandes para darte una prueba de la programación en el mundo real. En order de aparción trabajaremos en simulación de vida artificial, un lenguaje de programación, un juego de plataforma, un programa de pintura, y un sitio dinámico.

La parte del lenguaje del libro empieza con cuatro capítulos para presentar la estructura básica de JavaScript. Estos presentan estructuras de control (como la palabra `while` que viste en esta introducción), funciones(escribir tus propias operaciones), y estructuras de datos. Despeués de esto, serás capaz de escribir programas simples. Después, los capítulos 5 y 6 presentan técnicas para usar funciones y objetos para escribir código más *abstracto* y de esta manera mantener a la complejidad bajo control.

Después de un primer capítulo de proyecto, la primera parte del libro continúa con capítulos acerca de manejo y correccón de errores, expresiones regulares(una herramienta importanate para el manejo de datos de texto), y modularidad, otra arma contra la complejidad. El segundo capítulo de proyecto termina con la primera parte del libro.

En la segunda parte, los Capítulos 12 a 19, describen las herramientas a las que JavaScript tiene acceso en el navegador web. Aprenderás a mostrar cosas en la pantalla (Capítulos 13 y 16), repsonder a los datos de entrada del usuario (Capítulos 14 y 18), y a comunicarte a través de la red (Capítulo 17). Otra vez, hay dos proyectos en esta parte.

Después de eso, el Capítulo 20 describe Node.js, y en el Capítulo 21 se construye un sencillo sistema web usando esa herramienta.

Finalmente, el Capítulo 22 describe algunas de las consideraciones que se deben tener al optimizar programas de JavaScript para que sean rápidos.

Convenciones Tipográficas

En este libro,el texto escrito en fuente `monoespacio` representará elementos de programas; algunas veces programas completos y otras, partes de programas que hayan sido definidos cerca de ahí. Los programas(de los cuales has visto unos pocos), están escritos como sigue:

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

Algunas veces, para mostrar la salida que un programa produce, la salida esperada será escrita después de este, con dos diagonales y una flecha enfrente.

```
console.log(fac(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 separan y reagrupan. Las formas en el monitor no son más que ondas en el agua. La escencia permanece invisible debajo. Master Yuan-Ma, The Book of Programming

En el mundo de las computadoras sólo existen los datos. Puedes leer, modificar y crear nuevos datos, pero cualquier cosa que no sea datos simplemente no existe. Todos estos datos son guardados en largas secuencias de bits y por lo tanto son parecidos.

Los bits son cualquier tipo de cosas con dos valores, normalmente descritos como ceros y unos. Dentro de la computadora, toman formas como alta o baja carga eléctrica, una señal débil o fuerte, o un punto brillante u opaco en la superficie de un CD. Cualquier pieza de información discreta puede ser reducida a una secuencia de ceros y unos y por lo tanto representada como bits.

Por ejemplo, piensa cómo podrías representar el número 13 en bits. Funciona de la misma forma en que escribes números decimales, pero en vez de tener 10 dígitos, tienes sólo 2, y el peso de cada uno se incrementa por un factor de 2 de derecha a izquierda. Aquí están los bits que conforman el número 13, con los pesos de cada uno mostrados debajo de ellos.

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

Así que ese es el número binario 00001101, u 8 + 4 + 1, que equivale a 13.

Valores

Imagina un mar de bits. Un océano de estos. Una computadora moderna típica tiene más de 30 mil millones de bits en su almacenamiento de datos volátil. El almacenamiento no volátil (el disco duro o su equivalente) tiene un par de órdenes de magnitud más todavía.

Para ser capaz de trabajar con semejantes cantidades de bits sin perderte, puedes separarlos en pedazos que representen piezas de información. En un entorno en JavaScript, esos pedazos son llamados *valores*. Aunque todos los valores están hechos de bits, juegan diferentes roles. Cada valor tiene un tipo que determina su rol. Existen seis tipos básicos de valores en JavaScript: números, cadenas, Booleanos, objetos, funciones, y valores indefinidos.

Para crear un valor, sólo tienes que invocar su nombre. Esto es conveniente. No tienes que reuinir el material de construcción de tus valores o pagar por ellos. Sólo llamas uno y *woosh*, lo tienes. No son creados de la nada, por supuesto. Cada valor tiene que estar almacenado en algún lugar, y si quieres usar una cantidad enorme de estos al mismo tiempo, te podrías quedar sin bits. Afortunadamente, esto se convierte en un problema sólamente si los necesitas todos al mismo tiempo. Tan pronto como dejes de usar un valor se disipará, dejando sus bits para que sean reciclados como material de construcción de la próxima generación de valores.

Este capítulo introduce los elementos atómicos de los programas en JavaScript, los tipos de valores simples y los operadores que pueden actuar sobre tales valores.

Números

Los valores de tipo *número*(number) son, sin sorpresa alguna, valores numéricos. En un programa en JavaScript, se escriben de la siguiente forma:

```
13
```

Usa eso en un programa y causará que el patrón de bits para el número 13 exista dentro de la memoria de la computadora.

JavaScript usa una cantidad fija de bits, 64, para guardar un valor del tipo número. Existe un límite en la cantidad de patrones que se pueden hacer con 64 bits, lo que significa que la cantidad de números que puedes representar también es limitada. Para *N* dígitos

decimales, la cantidad de números que pueden ser representados es 10^*N*^. Similarmente, dados 64 dígitos binarios, puedes representar 2^64^ números diferentes, que es cerca de 18 cuatrillones (un 18 con 18 ceros después). Eso es mucho.

La memoria de la computadora solía ser mucho más pequeña, y la gente tendía a usar grupos de 8 ó 16 bits para representar sus números. Era fácil *desbordar* accidentalmente números tan pequeños: terminar con un número que no pudiera ser almacenado en el número dado de bits. Hoy, incluso las computadoras personales tienen mucha memoria, así que eres libre de usar grupos de 64 bits, lo que significa que necesitas preocuparte del desbordamiento sólo cuando estés tratando con números verdaderamente astronómicos.

No todos los número enteros debajo de 18 cuatrillones caben en un número de JavaScript. Esos bits también guardan números negativos, así que un bit indica el signo del número. Un problema mayor es que los números no enteros también deben ser representados. Para hacer esto, algunos de los bits son usados para guardar la posición del punto decimal. El número entero máximo real que puede ser guardado está más cerca del rango de los 9 trillones (15 ceros), que aún es satisfactoriamente grande.

Los números fraccionarios son escritos usando un punto.

```
9.81
```

Para números muy grandes o muy pequeños, también se puede usar la notación científica, añadiendo una "e" de "exponente", seguido por el exponente del número:

```
2.998e8
```

Esto es 2.998 × 10^8 = 299,800,000.

Cálculos con números enteros (en inglés llamados *integer*) más pequeños que el supracitado 9 trillones, están garantizados para siempre ser precisos. Desafortunadamente, cálculos con números fraccionarios generalmente no lo son. Justo como π (pi) no puede ser expresado precisamente por un número finito de dígitos decimales, muchos números pierden algo de precisión cuando sólo hay 64 bits disponibles para guardarlos. Esto es una pena, pero causa problemas prácticos sólo en algunas situaciones específicas. Lo importante es estar al tanto de esto y tratar a los números digitales fraccionarios como aproximaciones y no como valores precisos.

Aritmética

Lo principal que se hace con los números es la aritmética. Las operaciones aritméticas como la suma o la multiplicación toman dos valores y producen uno nuevo a partir de estos. Así es como lucen en JavaScript:

```
100 + 4 * 11
```

Los símbolos `+` y `*` son llamados *operadores*. El primero representa la suma y el segundo la multiplicación. Al poner un operador entre dos valores, se aplicará la operación a esos valores y producirá un nuevo valor.

¿Significa el ejemplo anterior: "suma 4 y 100, y multiplica el resultado por 11", o es la multiplicación ralizada antes de hacer la suma? Como pudiste haber adivinado, la multiplicación ocurre primero. Pero como en matemáticas, puedes cambiar esto mediante encerrar en paréntesis la suma.

```
(100 + 4) * 11
```

Para la resta existe el operador `-`, y la división se puede hacer con el operador `/`.

Cuando los operadores aparecen juntos sin paréntesis, el orden en el que son aplicados es determinado por su *precedencia*. El ejemplo muestra que la multiplicación se aplica antes que la suma. El operador `/` tiene la misma precedencia que `*`. De igual forma pasa con `+` y `-`. Cuando varios operadores con la misma precedencia aparecen juntos, como en `1 - 2 + 1`, son aplicados de izquierda a derecha: `(1 - 2) + 1`.

Estas reglas de precedencia son algo de lo que no te deberías de preocupar. Cuando tengas duda, simplemente agrega paréntesis.

Existe un operador aritmético más, que podrías no reconocer inmediatamente. El símbolo `%` es usado para representar la operación *sobrante*. `X % Y` es el sobrante de dividir `X` entre `Y`. Por ejemplo, `314 % 100` produce `14`, y `144 % 12` da `0`. La precedencia del sobrante es la misma que la de la multiplicació y división. Verás a menudo este operador referido como *módulo*, aunque técnicamente *sobrante* es más preciso.

Números Especiales

Hay tres valores especiales en JavaScript que son considerados números pero no se comportan como números normales.

Los primeros dos son `Infinity` y `-Infinity`, que representan infinitos positivos y negativos. `Infinity - 1` sigue siendo `Infinity`, y así por el estilo. No confíes mucho en los cálculos basados en infinitos. No son matemáticamente confiables y pronto te llevarán al

próximo número especial: `NaN`.

`NaN` son las siglas de “not a number” ("no es un número"), aunque es un valor del tipo número. Obtendrás este resultado cuando, por ejemplo, trates de calcular `0 / 0` (cero entre cero), `Infinity - Infinity`, o cualquier otra operación numérica que no produzca un resultado preciso, significativo.

Cadenas

El siguiente tipo de dato básico son las *cadenas*. Estas son usadas para representar texto. Son declaradas al poner el contenido entre comillas.

```
"Parcha mi bote con goma de mascar"
'Monkeys wave goodbye'
```

Tanto las comillas simples como las dobles pueden ser usadas para declarar cadenas de texto mientras coincidan al principio y al final.

Casi cualquier cosa puede estar entre comillas, y JavaScript creará una cadena. Pero unos cuantos caracteres son un poco difíciles. Puedes imaginar que poner comillas dentro de comillas puede ser difícil. *Newlines* (el carácter salto de línea, lo que obtines cuando presionas Enter), tampoco puede ser introducido entre comillas. La cadena tiene que permanecer en una sola línea.

Para hacer posible la inclusión de estos caracteres en una cadena de texto, la siguiente notación es usada: cuando una diagonal invertida(*backslash*: `\`) se encuentra dentro de un texto entre comillas, indica que el carácter siguiente tiene un significado especial. Esto es llamado *escapar* el carácter. Una comilla que es precedida por una diagonal invertida no terminará la cadena, sino que será parte de ella. Cuando un carácter `n` sigue a una diagonal invertida, se interpreta como una nueva línea. Similarmente, un `t` después de la diagonal invertida significa un tabulador. Tomemos la siguiente cadena:

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

El verdadero texto contenido es:

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

Existen, por supuesto, situaciones en las que querrás que una diagonal invertida sea sólo eso en una cadena de texto, no un código especial. Si dos diagonales invertidas están juntas, se volverán una, y sólo eso quedará como resultado en el valor de la cadena. Así es

como la cadena '*"Un carácter de nueva línea es escrito \"\n\"."* puede ser expresada:

```
"Un carácter de nueva línea es escrito \"\\n\"."
```

Las cadenas de texto no pueden ser divididas numéricamente, multiplicadas, o restadas, pero el carácter `+` *puede* ser usado en ellas. No suma, sino que *concatena*; pega dos cadenas. La siguiente línea produce la cadena `"concatenar"`:

```
"con" + "cat" + "e" + "nar"
```

Hay más maneras de manipular las cadenas, de las que hablaremos cuando lleguemos a los métodos en el link:04_data.html#methods[Capítulo 4].

Operadores Unitario

No todos los operadores son símbolos. Algunos son escritos como palabras. Un ejemplo es el operador `typeof`, que produce una cadena de texto que representa el tipo del valor que le pasaste.

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

Usaremos `console.log` para indicar que queremos ver el resultado de la evaluación de algo. Cuando corres ese código, el valor producido debería mostrarse en pantalla, aunque la forma en que aparece dependerá del entorno en que estés corriendo el programa.

Los otros operadores que hemos visto operaban sobre dos valores, pero `typeof` sólamente toma uno. Los operadores que usan dos valores son llamados operadores *binarios*, mientras que aquellos que sólo toman uno son llamados operadores *unitarios*. El operador menos puede usar tanto como operador binario como operador unitario.

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

Valores Booleanos

A menudo, necesitarás un valor que simplemente distinga entre dos posibilidades, como "sí" y "no" o "encendido" y "apagado". Para esto, JavaScript tiene un tipo *Booleano*, que tiene sólo dos valores, verdadero (true) y falso (false), que son simplemente estas palabras en

inglés.

Comparaciones

Esta es una forma de producir valores booleanos:

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

Los signos `>` y `<` son los símbolos tradicionales para "es mayor que" y "es menor que", respectivamente. Estos son operadores binarios. Aplicarlos resulta en un valor Booleano que indica si son ciertos en ese caso.

Las cadenas de texto pueden ser comparadas de la misma manera.

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

La manera en que las cadenas son ordenadas es más o menos alfabética: las letras mayúsculas son siempre "menores" que las minúsculas, así que `"Z" < "a"` es verdad, y los caracteres no alfabéticos (!, -, y así por el estilo) son también incluidos en el ordenamiento. La comparación real está basada en el estándar *Unicode*. Este estándar asigna un número a virtualemente cada carácter que alguna vez necesitarás, incluyendo caracteres del griego, árabe, japonés, tamil, y otros alfabetos. Tener tales números es útil para guardar las cadenas de texto dentro de la computadora porque hace posible representarlas como una secuencia de números. Cuando comparamos cadenas, JavaScript va de izquierda a derecha, comparando los códigos numéricos de los caracteres uno por uno.

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

```
console.log("Itchy" != "Scratchy")
// → true
```

Sólo esxiste un valor en JavaScript que no es igual a sí mismo, y este es `NaN`, que significa "no es un número".

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

La intención de `NaN` es representar el resultado de un cálculo sin sentido y como tal, no es igual al resultado de cualquier *otro* cálculo sin sentido.

Operadores Lógicos

Hay también algunas operaciones que pueden ser aplicadas a los valores Booleanos. JavaScript soporta tres operadores lógicos: *and*, *or* y *not*. Estos pueden ser usados para "razonar" con los Booleanos.

El operador `&&` representa la operación lógica *and* ("y"). Es un operador binario, y su resultado es verdadero(true) sólo si los dos valores dados son verdaderos.

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

El operador `||` denota la operación lógica *or* ("o"). Devuelve verdadero si cualquiera de los dos valores dados es verdadero.

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

Not (Negación) es escrito como un símbolo de admiración (`!`). Es un operador binario que voltea el valor que se le de; `!true` produce `false` y `!false` regresa `true`.

Cuando mezclamos estos operadores Booleanos con aritmética y otros operadores, no es siempre obvio cuándo se necesitan los paréntesis. En la prácitca, puedes avanzar sabiendo que de los operadores que hemos visto hasta ahora, `||` tiene la menor precedencia, después viene el `&&`, siguen los operadores de comparación(`>`, `==`, etc.), y después los demás operadores. Este orden ha sido elegido tal que, en expresiones típicas con la siguiente, sean necesarios tan pocos paréntesis como sea posible:

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

El último operador lógico del que hablaré no es unitario, ni binario, sino *ternario*, opera en tres valores. Este es escrito con un símbolo de interrogación y dos puntos, como sigue:

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

Este es llamado el operador *condicional* (o algunas veces el operador *tenario* dado que es el único operador de este tipo en el lenguaje). El valor a la izquierda del signo de interrogación "escoge" cuál de los otros dos valores resultará. Cuando es verdadero, el valor central es escogido, y cuando es falso, el valor de la derecha se da como resultado.

Valores Indefinidos (Undefined)

Existen dos valores especiales, escritos `null` y `undefined`, que son usados para denotar la ausencia de un valor con significado. Son valores en sí mismos, pero no poseen ninguna información.

Muchas operaciones en el lenguaje que no producen un valor con significado (lo verás después) producen `undefined` simplemente porque tienen que producir *algún* valor.

La diferencia en el significado entre `undefined` y `null` es un accidente del diseño de JavaScript, y no importa la mayoría del tiempo. En los casos en dónde realmente te tienes que preocupar de estos valores, te recomiendo tratarlos como intercambiables (más de esto en un momento).

Conversión automática de tipos

En la introducción, mencioné que JavaScript acepta casi cualquier programa que le des, incluso programas que hagan cosas raras. Esto es muy bien demostrado por las siguientes expresiones:

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

Cuando un operador es aplicado al tipo "incorrecto" de valor, JavaScript convertirá silenciosamente el valor en el tipo de dato que espera, usando un conjunto de reglas que a menudo no son lo que tú quieres o esperas. Esto es llamado *coerción de tipo*. Así que el `null` en la primera expresión se vuelve `0`, y el `"5"` en la segunda expresión se convierte en `5` (de cadena a número). Aún así, en la tercera expresión, `+` intenta hacer concatenación de cadenas antes de suma numérica, así que el `1` es convertido en `"1"` (de número a cadena).

Cuando algo que no se corresponde con un número de manera obvia (como `"cinco"` o `undefined`) es convertido a un número, el valor resultante es `NaN`. Las siguientes operaciones aritméticas sobre `NaN` seguirán produciendo `NaN`, así que si te encuentras con uno de estos en un lugar inesperado, busca conversiones accidentales de tipo.

Cuando comparamos valores del mismo tipo usando `==`, la salida es fácil de predecir: deberías obtener verdadero cuando los dos valores sean el mismo, excepto en el caso de `NaN`. Pero cuando los tipos son diferentes, JavaScript usa un complicado y confuso conjunto de reglas para determinar qué hacer. En la mayoría de los casos, sólo trata de convertir uno de los valores al tipo de dato del otro valor. Sin embargo, cuando `null` o `undefined` están en cualquier lado de la operación, resulta verdadero sólo en el caso de que los dos lados sean `null` o `undefined`.

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

Este último comportamiento es útil a menudo. Cuando quieres probar si un valor tiene un significado real en vez de `null` o `undefined`, simplemente comparas contra `null` con el operador `==` (o `!=`).

¿Y si quieres probar si algo se refiere precisamente al valor `false`? Las reglas para convertir cadenas y números a Booleanos dicen que `0`, `NaN` y la cadena vacía (`""`) cuentan como `false`, mientras que todos los demás valores cuentan como `true`. Debido a esto, expresiones como `0 == false` y `"" == false` también son verdaderas. Para casos como este en el que *no* quieres que ocurra ninguna conversión automática de tipos, existen dos operadores extra: `===` y `!==`. El primero prueba si un valor es precisamente igual a otro, y el segundo si no es precisamente igual. Así que `"" === false` es falso como se espera.

Yo recomiendo usar la comparación de tres caracteres defensivamente para evitar que conversiones de tipos inesperadas te causen problemas. Pero cuando estás seguro de que los tipos en ambos lados serán los mismos, no hay problema con usar los operadores más cortos.

Corto circuito de operadores lógicos

Los operadores lógicos `&&` and `||` manejan los valores de diferentes tipos de un modo peculiar. Convertirán el valor de su lado izquierdo para decidir qué hacer, pero dependiendo del operador y del resultado de la conversión, devuelven el valor del lado izquierdo *original* o el del lado derecho.

El operador `||`, por ejemplo, regresará el valor de la izquierda cuando pueda ser convertido a `true` y regresará el valor a la derecha en cualquier otro caso. Esta conversión funciona como esperarías con valores Booleanos y debería hacer algo análogo con valores de otros tipos.

```
console.log(null || "user")
// → user
console.log("Karl" || "user")
// → Karl
```

Esta funcionalidad permite al operador `||` ser usado como una manera de respaldarnos con un valor por defecto. Si le das en el lado izquierdo una expresión que puede producir un valor vacío, el valor de la derecha será usado como reemplazo dado el caso.

El operador `&&` funciona de manera similar, pero en sentido opuesto. Cuando el valor a su izquierda es algo que se convierte en falso, regresa este valor, y en otro caso regresa el valor de la derecha.

Otra importante propiedad de estos dos operadores es que el lado derecho sólo es evaluado cuando es necesario. En el caso de `true || X`, no importa lo que sea `X`, incluso en si es una expresión que hace algo **terrible**, el resultado será verdadero, y `X` no será evaluado nunca. Lo mismo ocurre para `false && X`, lo cuál es falso e ignorará `X`. Esto es llamado *evaluación de corto circuito* (short-circuit evaluation).

El operador condicional trabaja de una forma similar. La primera expresión es evaluada siempre, pero el segundo o tercer valor, el que no sea escogido, no se evalúa.

Resumen

Vimos cuatro tipos de valores de JavaScript en este capítulo: números, cadenas, Booleanos, y valores indefinidos.

Estos valores son creados al escribir su nombre (`true`, `null`) o valor (`13`, `"abc"`). Puedes combinar y transformar valores con los operadores. Vimos operadores binarios para aritmética(`+`, `-`, `*`, `/` y `%`), concatenación de cadenas (`+`), comparación (`==`, `!=`, `===`, `!==`, `<`, `>`, `<=`, `>=`), y lógica (`&&`, `||`), así como varios operadores unitarios

(`-` para hacer negativo un número, `!` para negar lógicamente y `typeof` para obtener el tipo de un valor) y un operador ternario (`?:`) para elegir entre dos valores basándonos en un tercero.

Esto te brinda suficiente información para usar JavaScript como una pequeña calculadora, pero no para mucho más. El siguiente capítulo empezará a entremezclar estas expresiones en programas básicos.

Estructura del Programa

> Y mi corazón brilla rojo debajo de mi delgada, translúcida piel y tienen que administrarme 10cc de JavaScript para hacerme regresar (respondo bien a las toxinas en la sangre). ¡Hombre, esa cosa te sacará de tus casillas! _why, Why's (Poignant) Guide to Ruby

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

Expresiones y declaraciones

En el Capítulo 1, creamos algunos valores y después les aplicamos operadores para obtener nuevos valores. Crear valores de esta forma es una parte esencial de cada programa de JavaScript, pero esto es sólo una parte.

Un fragmento de código que produce un valor es llamado una *expresión*. Cada valor que se escribe literalmente (tal como `22` o `"psicoanálisis"`) es una expresión. Una expresión entre paréntesis es también una expresión, como un operador binario aplicado a dos expresiones o un operador unario aplicado a una expresion.

Esto muestra parte de la belleza de una interfaz basada en el lenguaje. Las expresiones se pueden anidar en una forma muy similar a la forma de sub-frases en la que las lenguas humanas son anidadas, y las sub-frases pueden contener sus propias sub-frases, etc. Esto nos permite combinar expresiones para expresar cálculos arbitrariamente complejos.

Si una expresión corresponde a un fragmento de frase, una *declaración* en JavaScript corresponde a una frase completa en lenguaje humano. Un programa es simplemente una lista de declaraciones.

El tipo más simple de declaración es una expresión con un punto y coma después de ella. Esto es un programa:

```
1;
!false;
```

Es un programa inútil, sin embargo. Una *expresión* puede estar presente para sólo producir un valor, que puede entonces ser utilizado por la expresión que la contiene. Una *declaración* existe por sí sola y que equivale a algo sólo si afecta al mundo. Podría mostrar algo en la pantalla -que cuenta como cambiar el *mundo* o podría cambiar el estado interno de la

máquina de estados de manera que afectará las declaraciones que vienen despues de ella. Estos cambios se llaman *efectos colaterales*. Las declaraciones en el ejemplo anterior solo producen los valores `1` y `verdadero` y los desechan inmediatamente. Esto no deja ningún cambio en el mundo en absoluto. Al ejecutar el programa, nada observable sucede.

En algunos casos, JavaScript te permite omitir el punto y coma al final de una declaración. En otros casos, tiene que estar allí, o la siguiente linea será tratada como parte de la misma declaración. Las reglas para cuando se puede omitir con seguridad son algo complejas y propensas a errores. En este libro, cada declaración que necesite un punto y coma siempre será terminada por un punto y coma. Te recomiendo que hagas lo mismo en tus propios programas, al menos hasta que hayas aprendido más sobre sutilezas involucradas en omitir el punto y coma.

Variables

¿Cómo mantiene un programa su estado interno? ¿Cómo recuerda algo? Hemos visto cómo producir nuevos valores de viejos valores, pero esto no cambia los valores antiguos, y el nuevo valor tiene que ser inmediatamente utilizado o se disipará de nuevo. Para atrapar y mantener los valores, JavaScript proporciona una cosa llamada *variable*.

```
var atrapado = 5 * 5;
```

Y eso nos da nuestra segunda clase de declaración. La palabra especial(*palabra clave* o *keyword*) `var` indica que esta frase va a definir una variable. Es seguida por el nombre de la variable y, si queremos dar de inmediato un valor, con un operador de `=` y una expresión.

La declaración anterior crea una variable llamada `atrapado` y se usa para retener el número que se produce al multiplicar 5 por 5.

Después de que una variable se ha definido, su nombre puede ser usado como una expresión. El valor de esa expresión es el valor que la variable alberga actualmente. He aquí un ejemplo:

```
var diez = 10;
console.log(diez * diez);
// → 100
```

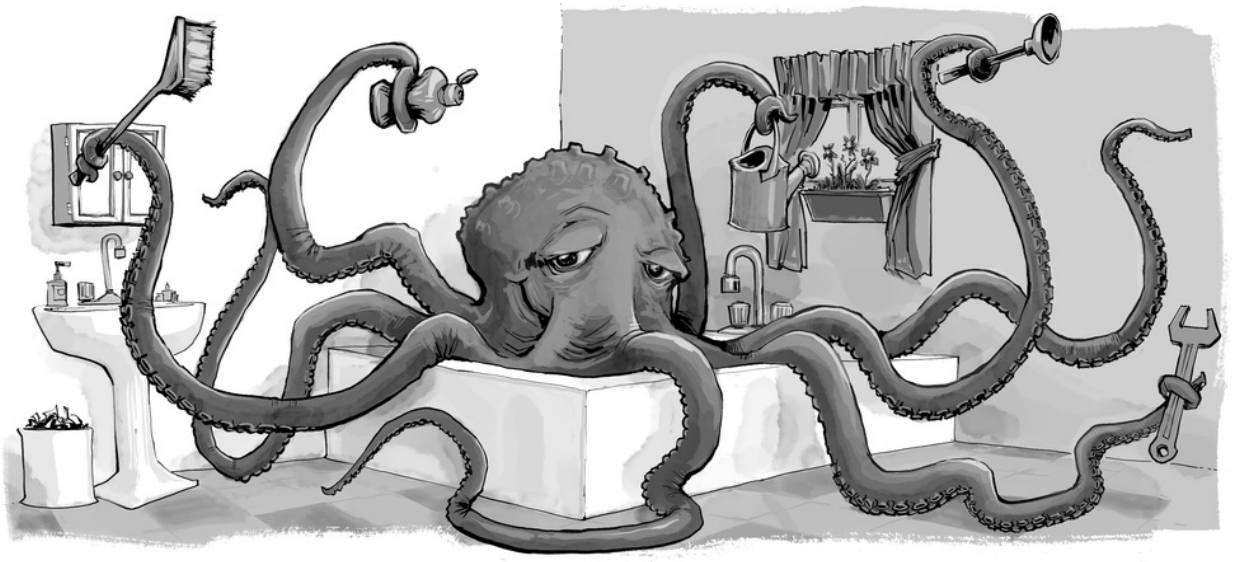
Los nombres de variables pueden ser cualquier palabra que no sea una palabra clave (tal como `var`). Estos no pueden incluir espacios. Los dígitos también pueden ser parte de la variable nombre — `catch22` es un nombre válido, por ejemplo-, pero el nombre no debe

comenzar con un dígito. Un nombre de variable no puede incluir puntuación, a excepción de los caracteres `$` y `_`.

Cuando una variable apunta a un valor, eso no quiere decir que está ligada a ese valor para siempre. El operador `=` se puede utilizar en cualquier momento en variables existentes para desconectarlas de su valor actual y apuntarlas a uno nuevo.

```
var tono = "claro";
console.log(tono);
// → claro
tono = "oscuro";
console.log(tono);
// → oscuro
```

Podrías imaginar las variables como tentáculos, en lugar de la cajas. Estas no *contienen* valores; los *agarran*; dos variables pueden referirse al mismo valor. Un programa sólo puede acceder a los valores que todavía mantiene atrapados. Cuando necesitas recordar algo, haces crecer un tentáculo para agarrarlo o cambias unos de tus tentáculos existentes para agarrarlo.



Veamos un ejemplo. Para recordar el número de dólares que Luigi aún te debe, se crea una variable. Y luego, cuando te paga $35, le das a esta variable un valor nuevo.

```
var deudaDeLuigi = 140;
deudaDeLuigi = deudaDeLuigi - 35;
console.log(deudaDeLuigi);
// → 105
```

Cuando se define una variable sin darle un valor, el tentáculo no tiene nada que sostener, por lo que termina en el aire. Si preguntas por el valor de una variable vacía, obtendrás el valor `undefined` (indefinido).

Una sola declaración `var` puede definir múltiples variables. Las definiciones deben estar separadas por comas.

```
var uno = 1, dos = 2;
console.log(uno + dos);
// → 3
```

Palabras clave y palabras reservadas

Palabras con un significado especial, como `var` , son *palabras clave*, y no pueden ser utilizadas como nombres de variables. También hay un número de palabras que son “reservadas para uso” en *futuras* versiones de JavaScript. Estas también están oficialmente no permitidas como nombres de variables, aunque algunos entornos de JavaScript las permiten. 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 in
instanceof interface let new null package private
protected public return static super switch this
throw true try typeof var void while with yield
```

No te preocupes por memorizarlas, pero recuerda que esto podría ser el problema cuando una definición de variable no funcione como se esperaba.

El entorno

La colección de variables y sus valores que existe en un momento dado se llama el *entorno*. Cuando un programa se pone en marcha, este entorno no está vacío. Siempre contiene variables que forman parte del lenguaje ((estándar)), y la mayoría del tiempo, contiene variables que proporcionan formas de interactuar con el sistema que lo contiene. Por ejemplo, en un *navegador*, existen variables y funciones para inspeccionar e influir en la página web cargada en ese momento y leer entrada del *ratón* y del *teclado*.

Funciones

Una gran cantidad de los valores proporcionados en el entorno por defecto tienen el tipo *function*. Una función(function) es un pedazo de programa encerrado en un valor. Tales valores pueden ser *aplicados* con el fin de ejecutar el programa envuelto. Por ejemplo, en un entorno de navegador, la variable `alert` contiene una función que muestra un pequeño cuadro de diálogo con un mensaje. Se utiliza como sigue:

```
alert("¡Good Morning!");
```

La ejecución de una función es denominada *invocar*, *llamar*, o *aplicar* la función. Puedes llamar a una función poniendo *paréntesis* después de una expresión que produce un valor de la función. Por lo general, se usa directamente el nombre de la variable que contiene la función. Los valores entre paréntesis se le pasan a el programa dentro de la función. En el ejemplo, la función `alert` utiliza la cadena que le damos como el texto que se mostrará en el cuadro de diálogo. Los valores dados a las funciones se denominan *argumentos*. La función `alert` necesita solo uno, pero otras funciones pueden necesitar un número diferente o diferentes tipos de argumentos.

La función console.log

La función `alert` puede ser útil para imprimir cuando estamos experimentando, pero quitar del camino todas esas pequeñas ventanas puede desesperarte. En ejemplos pasados, hemos usado `console.log` para devolver valores. La mayoría sistemas JavaScript (incluyendo a todos los navegadores web modernos y a Node.js) nos dan una función `console.log` que imprime sus argumentos en *algún* dispositivo de salida de texto. En los navegadores la salida queda en la consola de JavaScript. Esta parte del navegador está escondida por defecto, pero la mayoría de los navegadores la abren cuando presionas F12 o, en Mac, cuando presionas Command-Option-I. Si esto no funciona, busca en los menús un item llamado "Consola Web" o "Herramientas de Desarrollador".

Cuando corras los ejemplos, o tu propio código, en las páginas de este libro, la salida de `console.log` será mostrada después del ejemplo, en vez de en la consola de JavaScript del navegador.

```
var x = 30;
console.log("el valor de x es", x);
// → el valor de x es 30
```

Aunque los nombres de variable no pueden contener el caracter punto, `console.log` claramente tiene uno. Esto es porque `console.log` no es una variable simple. Es en realidad una expresión que trae la propiedad `log` del valor mantenido por la variable `console`. Veremos que significa exactamente en el Capítulo 4.

Valores de Retorno

Mostrar un cuadro de diálogo o escribir texto en la pantalla es un *efecto secundario*. Muchas funciones son útiles porque producen valores, y en ese caso, no necesitan tener un efecto secundario para ser útiles. Por ejemplo, la función `Math.max` toma un número indeterminado de números y regresa el más grande.

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

Cuando una función produce un valor, se dice que *regresa* ese valor. Cualquier cosa que produce un valor es una expresión en JavaScript, lo que significa que puede ser usada dentro de expresiones más grandes. Aquí, una llamada a `Math.min`, que es lo opuesto a `Math.max`, es usada como entrada de un operador de suma:

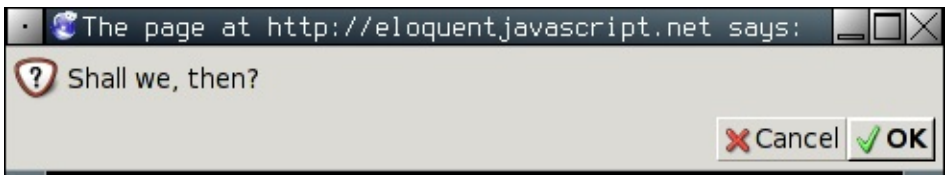
```
console.log(Math.min(2, 4) + 100);
// → 102
```

El próximo capítulo explica como escribir tus propias funciones.

Pedir información y confirmar

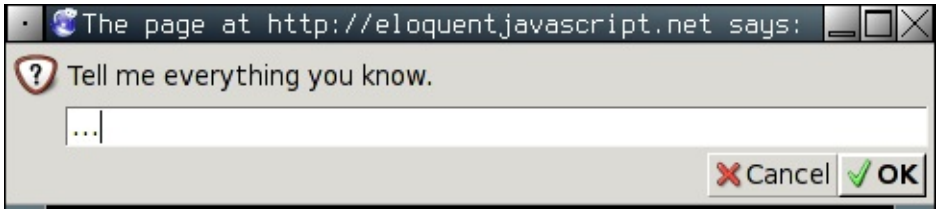
Los entornos de navegador tienen otras funciones más allá de `alert` para mostrar ventanas. Puedes preguntar al usuario una cuestión estilo OK/Cancelar usando `confirm`. Esto regresa un Booleano: `true` si el usuario hace click en OK y `false` si el usuario presiona en Cancelar.

```
confirm("¿Entonces, deberíamos?");
```



La función `prompt` puede ser usada para hacer una pregunta "abierta". El primer argumento es la pregunta, el segundo es un texto con el que el usuario inicia. Se puede escribir una línea de texto en el cuadro de diálogo, y la función regresará este texto como una cadena.

```
prompt("Tell me everything you know.", "...");
```



Estas dos funciones no son usadas mucho en la programación web moderna, principalmente porque no tienes control sobre la forma en que las ventanas resultantes se verán, pero son útiles para programas de prueba y experimentos.

Control de flujo

Cuando tu programa contiene más de una sentencia, las sentencias son ejecutadas (fácil de predecir), de arriba hacia abajo. Como un ejemplo básico, este programa tiene dos sentencias. La primera le pide un número al usuario, y la segunda, que se ejecuta después, muestra el _cuadrado_de ese número

```
var elNumero = Number(prompt("Dame un número", ""));
alert("Tú 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 cadena (string), y queremos un número. Hay funciones similares llamadas `String` y `Boolean` que convierten valores a estos tipos.

Aquí está la trivial representación esquemática de un flujo de control recto.

Ejecución Condicional

Ejecutar sentencia en línea recta no es la íunica opción que tenemos. Una alternativa es la *ejecución condicional*, en dondne escogemos entre dos rutas diferentes basados en un valor Booleano, como el siguiente:

La ejecución condicional se escribe con la palabra clave `if` en JavaScript. En el caso sencillo, queremos que algo de código se ejecute si, y sólo si, cierta condición se cumple. Por ejemplo, en el programa previo, podríamos querer mostrar el cuadrado de la entrada sólo si la entrada es un número.

```
var elNumero = Number(prompt("Dame un número", ""));
if (!isNaN(elNumero))
  alert("Tu número es la raíz cuadrada de " + elNumero * elNumero);
```

Con esta modificación, si le das "queso", no se mostrará ninguna salida.

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

La función `isNaN` es una función estándar de JavaScript que regresa `true` sólo si el argumento que le diste es `NaN`. Resulta que la función Number regresa `NaN` cuando le das una cadena que no “a menos que `elNumero` no sea un número”.

A menudo no sólo tendrás código que se ejecute cuando una condición sea verdadera, sino también que maneje el otro caso. Este camino alternativo es representado por la segunda flecha en el diagrama. La palabra clave `else` puede ser usada, junto con `if`, para crear dos rutas de ejecución separadas y alternativas.

```
var elNumero = Number(prompt("Dame un número", ""));
if (!isNaN(elNumero))
  alert("Tu número es la raíz cuadrada de " + elNumero * elNumero);
else
  alert("Hey. ¿Por qué no me diste un número?");
```

Si tenemos más de dos caminos a escoger, varios pares de `if` / `else` pueden ser "encadenados". Aquí hay un ejemplo:

```
var num = Number(prompt("Dame un número", "0"));

if (num < 10)
  alert("Chico");
else if (num < 100)
  alert("Mediano");
else
  alert("Grande");
```

El programa primero checará sí `num` es menor que 10. Si lo es, escoge ese camino, muestra `"Chico"` y termina. Si no lo es, toma el camino `else`, que en sí mismo contiene un segundo `if`. Si la segunda condición (`< 100`) se cumple, significa que el número está entre 10 y 100, y se muestra `"Mediano"`. Si no lo es, el segundo y último `else` es escogido.

El diagrama de flujo para este programa es algo así:

bucles while y do

Piensa en un programa que imprima todos los números primos del 1 al 12. Una manera de escribirlo sería como sigue:

```
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 trabajar *menos*, no más. Si necesitamos todos los números menores que 1,000, lo anterior sería imposible de trabajar. Lo que necesitamos es una forma de repetir algo de código. Esta forma de control de flujo es llamada *bucle*:

El control de flujo del bucle nos permite ir atrás a algún punto en el programa donde estábamos antes y repetirlo con nuestro actual estado del programa. Si combinamos esto con una variable contadora, podemos hacer algo así:

```
var number = 0;
while (number <= 12) {
  console.log(number);
  number = number + 2;
}
// → 0
// → 2
//   … etcetera
```

Una sentencia que comienza con la palabra clave `while` crea un bucle. Después de la palabra while viene una expresión en paréntesis y después una sentencia, muy parecido a el `if` . El bucle ejecuta la sentencia mientras la expresión produzca un valor que sea `true` cuando se convierte a un tipo Booleano.

En este bucle, queremos tanto imprimir el número, como sumar dos a nuestra variable. Cuando necesitamos ejecutar múltiples sentencias dentro de un bucle, lo encerramos en llaves (`{` y `}`). Las llaves hacen por las sentencias lo que los paréntesis hacen por las expresiones: las agrupan, haciéndolos valer por una sola sentencia. Una secuencia de sentencias encerradas en llaves es llamada un *bloque*.

Muchos programadores de JavaScript encierran cada cuerpo de un bucle `if` en llaves. Lo hacen en nombre de la consistencia y para evitar tener que añadir o quitar las llaves cuando el número de sentencias en el cuerpo cambie. En este libro escribiré la mayoría de los cuerpos de una sola sentencia sin bloques, porque valúo la brevedad. Tú eres libre de usar el estilo que prefieras.

La variable `número` demuestra la forma en que una variable puede dar seguimiento al progreso de un programa. Cada vez que el bucle se repite, `numero` se incrementa en `2` . Entonces, al principio de cada repetición, es comparada con el número `12` para decidir si el programa ha hecho todo el trabajo que tenía que hacer.

Como un ejemplo que hace realmente algo útil, podemos escribir un programa que calcula y mustra el valor de 2^10 (dos a la décima potencia). Usamos dos varianles: una para mantener el resultado y una para contar cuantas veces hemos multiplicado este resultado por dos. El bucle verifica si la segunda variable ha llegado a 10 y entonces actualiza las dos variables.

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

El contador pudo también empezar en `1` y verificar que el contador sea `<=10`, pero, por razones que se aclararán en el Capítulo 4, es una buena idea acostumbrarse a contar desde 0.

El bucle `do` es una estructura de control similar al bucle `while`. Se diferencia en sólo un punto: un bucle `do` siempre ejecuta su cuerpo por lo menos una vez y empieza a verificar si debería parar sólo después de la primera ejecución. Para reflejar esto, la condición aparece después del cuerpo del bucle:

```
do {
  var yourName = prompt("Who are you?");
} while (!yourName);
console.log(yourName);
```

Este programa te obligará a introducir un nombre. Preguntará una y otra vez hasta que obtenga algo que no sea una cadena vacía. Aplicar el operador `!` convierte un valor a Booleano negándolo y todas las cadenas excepto `""` se convierten a `true`. Esto significa que el bucle continúa corriendo hasta que des un nombre que no sea una cadena vacía.

Indentando código

Probablemente has notado los espacios que pongo en frente de algunas sentencias. En JavaScript, no son requeridos, la computadora aceptara el programa bien sin ellos. De hecho, incluso los saltos de línea en los programas son opcionales. Puedes escribir un programa en una sola línea si así lo prefieres. El rol de la indentación dentro de los bloques es hacer notar la estructura del código. En código complejo, en dónde nuevos bloques son abiertos dentro de otros bloques, puede ser difícil de ver en dónde termina uno y empieza otro. Con la indentación correcta, la forma visual del programa se corresponde con la forma de los bloques dentro de él. A mí me gusta usar dos espacios por cada bloque abierto, pero los gustos varían; algunas personas usan cuatro espacios, y algunas otras usan el carácter `tab`.

Bucles for

Muchos bucles siguen el patrón de los ejemplos previos del `while`. Primero, una variable *“contador”* es creada para dar seguimiento al progreso del bucle. Entonces viene un bucle `while`, cuya expresión condicional normamelte verifica si el contador ha alcanzafo cierto límite. En el final del cuerpo del bucle, el contador es actualizado para dar seguimiento al progreso.

Debido a que este patrón es tan común, JavaScript y los lenguajes similares proveen una forma un poco más corta y facil de entender, el bucle `for`.

```
for (var number = 0; number <= 12; number = number + 2)
  console.log(number);
// → 0
// → 2
//   … etcetera
```

Este programa equivale exactamente al ejemplo anterior. El único cambio es que todas las *declaraciones* que se refieren al "estado" del bucle están ahora agrupadas entre ellas.

Los paréntesis posteriores a una palabra clave `for` deben contener dos punto y coma. La primera parte antes del primer punto y coma *inicializa* el bucle, por lo general definiendo la variable. La segunda parte es la expresión que *comprueba* si el bucle debe continuar. La parte final *actualiza* el estado del bucle después de cada repetición. La mayoría de las veces, esto es más corto y claro que una construcción `while`.

Aquí el código que computa 2^10, usando `for` en lugar de `while`:

```
var result = 1;
for (var counter = 0; counter < 10; counter = counter + 1)
  result = result * 2;
console.log(result);
// → 1024
```

Nota que aunque ningún bloque está abierto con `{`, la declaración en el bucle está aún sangrada con dos espacios para hacer claro que *"pertenece"* a la línea anterior.

Deteniendo un bucle

Hacer que la condicion pruduzca `false` en el bucle no es la única forma en que un bucle puede terminar. Hay una declaración especial llamado `break` que tiene el efecto de saltar inmediatamente fuera del bucle cerrado.

Este programa ilustra la sentencia `break`. Encuentra el primer número que es mayor o igual a 20 y divisible entre 7.

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

Usando el operador (`%`) es una forma fácil de comprobar si un número es divisible entre otro número. Si es así, el resto de su división es cero.

El constructor `for` en el ejemplo no tiene una parte que revise el fin del bucle. Esto significa que el bucle nunca se detendrá a menos que la declaración `break` se ejecute desde adentro.

Si tu dejaras fuera la declaración `break` o escribieras accidentalmente una condición que siempre produce verdadero, tu programa quedará atorado en un *bucle infinito*. Un programa atorado en un bucle infinito nunca terminará de correr, lo que normalmente es algo malo.

Si creas un bucle infinito en uno de los ejemplos de estas páginas, normalmente el navegador te preguntara si quieres detener la secuencia de comandos después de unos cuantos segundos. Si eso falla, tendrás que cerrar la pestaña en la que estás trabajando, o en algunos navegadores cerrarlo completamente, con el fin de recuperarlo.

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

Actualizar variables en breve

Particularmente cuando se hace un bucle, un programa a menudo necesita "actualizar" una variable para mantener un valor basado en el valor previo de esa variable.

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

JavaScript provee un atajo a esto:

```
counter += 1;
```

Atajos similares trabajan para muchos otros operadores, como `resultado *= 2` para doblar el `resultado` o `contador -= 1` para contar descendente.

Esto nos permite acortar un poco más nuestro ejemplo de conteo.

```
for (var number = 0; number <= 12; number += 2)
  console.log(number);
```

Para `contador += 1` y `contador -= 1`, existen incluso equivalentes más cortos: `contador++` y `contador--`.

Enviando en un valor con switch

Es común que el codigo se vea así:

```
if (variable == "value1") action1();
else if (variable == "value2") action2();
else if (variable == "value3") action3();
else defaultAction();
```

Hay un constructor llamado switch que está pensado para funcionar como un "despachador" en una manera más directa. Desafortunadamente, la sintaxis que JavaScript usa para esto (lo cual hereda de lenguajes de programación como C/Java) que es de alguna manera menos eficiente que una cadena de declaraciones if que a menudo se ve mejor. Aquí un ejemplo:

```
switch (prompt("What is the weather like?")) {
  case "rainy":
    console.log("Remember to bring an umbrella.");
    break;
  case "sunny":
    console.log("Dress lightly.");
  case "cloudy":
    console.log("Go outside.");
    break;
  default:
    console.log("Unknown weather type!");
    break;
}
```

Podrás poner cualquier número de etiquetas `case` dentro del bloque abierto por `switch`. El programa saltará a la etiqueta que corresponde al valor que switch dió o el default si no coincide el valor. Esto empieza a ejecutar declaraciones allí, aunque estén bajo otra etiqueta, hasta que alcancen una declaración break. En algunos casos, como el del caso "sunny" en el ejemplo, esto puede ser usado para compartir algo de código entre casos (esto recomienda ir fuera por ambos climas soleado y nuboso). Pero ten cuidado: es fácil olvidar un break`, lo cual causará que el programa ejecute código que no quieres que se ejecute.

Mayúsculas

Los nombres de variables no debe contener espacios, sin embargo es común ayudarse usando múltiples palabras que claramente describan lo que la variable representa. Estas son casi tus única opciones para escribir un nombre de variable con varias palabras en ella.

```
fuzzylittleturtle
fuzzy_little_turtle
FuzzyLittleTurtle
fuzzyLittleTurtle
```

El primer estilo puede ser difícil de leer. Personalmente, Me gusta como se ven los guiones bajos, sin embargo ese estilo es un poco complicado de escribir. Las funciones comunes JavaScript, y la mayoría de programadores JavaScript, siguen el último estilo - ellos ponen en mayúsculas cada palabra excepto la primera. No es difícil acostumbrase a cosas pequeñas como esta, y escribir código con estilos de nombres mixtos puede ser tedioso de leer, así que nosotros sólo seguiremos este convención.

En algunos casos, como en la función `Number`, la primera letra de una variable es también mayúscula. Esto se hizo para señalar esta función como un constructor. Lo que es un constructor vendrá claramente en el capítulo 6. Por ahora, lo importante es no estar preocupado por esta aparente falta de consistencia.

Comentarios

A menudo, el código crudo no transmite toda la información que quieres que un programa transmita a los lectores humanos, o transmite en una forma como encriptada que la gente no puede entender. En otros momentos, te puedes sentir poético o queriendo incluir algunos pensamientos como parte de tu programa. Esto es para lo que son los *comentarios*.

Un comentario es un pedazo de texto que es parte de un programa pero es completamente ignorado por la computadora. JavaScript tiene dos maneras de escribir comentarios. Para escribir un comentario de una sola línea, puedes usar dos diagonales (`//`) y el texto del comentario después.

```
var accountBalance = calculateBalance(account);
// It's a green hollow where a river sings
accountBalance.adjust();
// Madly catching white tatters in the grass.
var report = new Report();
// Where the sun on the proud mountain rings:
addToReport(accountBalance, report);
// It's a little valley, foaming like light in a glass.
```

Un comentario `//` va únicamente al final de la línea. Una sección de texto entre `/*` y `*/` será ignorada, sin tener en cuenta lo que contenga. Esto es a menudo útil para agregar bloques de información acerca de un archivo o un trozo de programa.

```
/*
 I first found this number scrawled on the back of one of
 my notebooks a few years ago. Since then, it has often
 dropped by, showing up in phone numbers and the serial
 numbers of products that I've bought. It obviously likes
 me, so I've decided to keep it.
*/
var myNumber = 11213;
```

Resumen

Ahora tu sabes que un programa está construido por declaraciones, las cuales por si mismas contienen más declaraciones. Las declaraciones tienden a contener expresiones, las cuales en si mismas pueden ser construidas desde expresiones más pequeñas.

Poner declaraciones después de otra, te da un programa que es ejecutado de arriba a abajo. Puedes ingresar interrupciones en el control de flujo utilizando declaraciones condicionales (if, else, y switch) y de bucle (while, do, y for).

Las variables pueden ser usadas para archivar pedazos de datos dentro de un nombre, y son útiles para rastrear estados en tu programa. El ambiente es el conjunto de variables que son definidas. Los sistemas JavaScript siempre ponen un número de variables estándar útiles a tu ambiente.

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

Ejercicios

Si no estás seguro de como probar tus soluciones a los ejercicios, dirigete a la introducción.

Cada ejercicio comienza con una descripción de un problema. Léelo y trata de resolver el ejercicio. Si te encuentras en problemas, considera leer los consejos después del ejercicio. Las soluciones completas a los ejercicios no están incluidas en este libro, pero puedes encontrarlas en línea en http://eloquentjavascript.net/code. Si quieres aprender algo de los ejercicios, recomiendo buscar las soluciones únicamente después de resolver el ejercicio, o al menos después de que hayas tratado lo más duro posible y que tengas un ligero dolor de cabeza.

Buclear un triangulo

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

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

Puede ser útil saber que puedes encontrar el largo de la cadena de texto escribiendo .lenght después de esta.

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

La mayoría de los ejercicios contienen un pedazo de código que puedes modificar para resolver el ejercicio. Recuerda que puedes clickear bloques de código para editarlos.

```
// Your code here.
```

!!Consejo!!

Puedes comenzar con un programa que simplemente imprima la salida de números 1 al 7, los cuales puedes derivar haciendo una pocas modificaciones al ejemplo de impresión de número par dado antes en este capítulo, donde el bucle `for` fue presentado.

Ahora considera la equivalencia entre números y cadenas de texto de caracteres de número. Puedes ir de 1 a 2 agregando 1 (`+= 1`). Puedes ir de `"#"` a `"##"` agregando el caracter (`+= "#"`). Por lo tanto, tu solución puede estar cerca del programa *number-printing*.

FizzBuzz

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

Cuando tengas trabajando eso, modifica tu programa para imprimir `"FizzBuzz"`, para números que son divisibles por ambos 3 y 5 (y que aún imprima `"Fizz"` o `"Buzz"` para números divisibles por únicamente uno de esos).

(Esto es en realidad una pregunta de entrevista que ha sido elaborada para quitar un significante porcentaje de candidatos a programador. Entonces si lo resuelves, puedes sentirte bien contigo mismo.)

```
// Your code here.
```

!!Consejo!!

Ir sobre los números es claramente un trabajo de bucle, y seleccionar que se imprime es una cuestión de ejecución condicional. Recuerda el truco de usar un operador de resultado (`%`) para revisar si un número es divisible por otro número (tiene un resultado cero).

En la primera versión, existen tres posibles resultados para cada número, entonces tienes que crear una cadena de `if` / `else if` / `else`.

La segunda versión del programa tiene un solución directa e inteligente. La manera simple es agregar otra "rama" para precisamente probar la condición dada. Para el método inteligente, construye una cadena de texto conteniendo la palabra o palabras a salir, e imprime ya sea esta palabra o el número si es que no hay una palabra, potencialmente haciendo uso del operador elegante `||`.

Tablero de ajedréz

Escribe un programa que cree una cadena de texto que represente una rejilla de 8x8, usando caracteres *newline* para separar líneas. A cada posición de la rejilla ya sea un espacio o un caracter "#". Los caracteres deberían formar un tablero de ajedrez.

Pasando esta cadena de texto a `console.log` debería mostrar algo como esto:

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

Cuando tienes un programa que genera este patrón, define una variable `size = 8` y cambia el programa de modo que este trabaje para cualquier tamaño, imprimiendo una rejilla de ancho y alto dado.

```
// Your code here.
```

!!Consejo!!

La cadena de texto puede ser construida comenzando con un (`""`) vacío y repetidamente agregando caracteres. Un caracter newline es escrito " `"\n"` .

Usa `console.log` para inspeccionar la salida de tu programa.

Para trabajar con dos dimensiones, necesitarás un bucle dentro de un bucle. Pon llaves alrededor de los cuerpos de ambos bucles para hacer fácil de ver donde empiezan y terminan. Trata correctamente de identificar los cuerpos. El orden de los bucles deben seguir el orden en cual construiremos la cadena de texto (línea a línea, izquierda a derecha, arriba a abajo). Entonces para que el bucle exterior maneje las líneas y el bucle interior maneje los caracteres en una línea.

Necesitas dos variables para rastrear tu progreso. Para saber si poner un espacio o un signo de número a una determinada posición, puedes probar si la suma de los dos contadores es par (`%2`).

Poniendo fin a una línea agregando un caracter newline sucede después que la línea ha sido construida, por lo tanto haz esto después del bucle pero dentro del otro bucle.

Funciones

> La gente piensa que las ciencias de la computación son el arte de los genios pero la realidad es lo contrario, es sólo un montón de gente haciendo cosas que se construyen sobre otras, como una pared de piedras muy pequeñas. Donald Knuth

Has visto valores función, como `alert`, y cómo llamarlos. Las funciones son el pan de cada día en la programación en JavaScript. El concepto de de envolver una porción del programa en un valor (variable) tiene muchos usos. Es una herramienta para estructurar programas más grandes, para reducir la repetición, para asociar nombres con subprogramas, y para separar estos programas de los demás.

La aplicación más obvia de las funciones es la de definir un nuevo vocabulario. Crear nuevas palabras en la prosa regular en lenguaje humano es usualmente un mal estilo. Pero en programación, es indispensable.

Un adulto promedio tiene unas 20,000 palabras en su vocabulario. Pocos lenguajes de programación tienen 20,000 comandos incorporados. Y el vocabulario que *está* disponible tiende a ser definido de forma más precisa, así que es menos flexible que en un lenguaje humano. En consecuencia, usualmente *tenemos* que añadir algo de nuestro propio vocabulario para evitar repetirnos demasiado.

Definiendo una función

Una definición de una función es sólo una definición regular de una *variable* cuando ocurre que el valor dado a la variable es una función. Por ejemplo, el siguiente código define la variable `cuadrado` para referirse a la función que produce el cuadrado de un número dado:

```
var cuadrado = function(x) {
  return x * x;
};

console.log(cuadrado(12));
// → 144
```

Una función es creada por una expresión que empieza con la palabra reservada `function`. Las funciones tienen un conjunto de *parametros* (en este caso sólo `x`) y un *cuerpo*, que contiene las sentencias que serán ejecutadas cuando la función sea llamada. El cuerpo de la función tiene que estar siempre encerrado en llaves, incluso cuando consista de una sola instrucción (como en el ejemplo previo).

Una función puede tener varios parámetros o puede no tener ninguno. En el siguiente ejemplo `hazRuido` no tiene parámetros, mientras que `potencia` tiene dos:

```
var hazRuido = function() {
  console.log("Pling!");
};

hazRuido();
// → Pling!

var potencia = function(base, exponente) {
  var resultado = 1;
  for (var cuenta = 0; cuenta < exponente; cuenta++)
    resultado *= base;
  return resultado;
};

console.log(potencia(2, 10));
// → 1024
```

Algunas funciones producen un valor, como `potencia` y `cuadrado`, y algunas no, como `hazRuido`, la cuál produce sólo un efecto secundario. Una sentencia `return` determina el valor que una función regresa. Cuando el control pasa a esta sentencia, inmediatamente salta fuera de la función actual y pasa el valor retornado a la código que llamó la función. La palabra reservada `return` sin una expresión después de ella hará que la función devuelva `undefined`.

Parámetros y ámbitos

Los *parametros* para una función se comportan como variables normales, pero su valor inicial esta dado por *quien llama* a la función, no por el código mismo de la función.

Un propiedad importante de las funciones es que las varibales creades dentro de ellas, incluyendo sus parámetros, son *locales* para la función. Esto significa, por ejemplo, que la varibale `resultado` en el ejemplo `potencia` será creada de nuvo cada vez que la función es llamada, y estas instancias separadas no interfieren entre ellas.

Esta "localidad" de las variables aplica sólo a los parámetros y variables declaradas con la palabra reservada `var` dentro del cuerpo de la función. Las variables declaradas fuera de cualquier función son llamadas *globales*, porque son visibles a través de todo el programa. Es posible acceder a estas variables desde dentro de una función, mientras no hayas declarado una variable local con el mismo nombre.

El siguiente código demuestra eso. Define y llama dos funciones que asignan un valor a la variable `x`. La primera declara la variable como local y de esta manera cambia únicamente la variable que creó. La segunda no declara `x` localmente, así que la `x` dentro de ella hace referencia a la `x` definida al principio del ejemplo.

```
var x = "fuera";

var f1 = function() {
  var x = "dentro de f1";
};
f1();
console.log(x);
// → fuera

var f2 = function() {
  x = "dentro de f2";
};
f2();
console.log(x);
// → dentro de f2
```

Este comportamiento ayuda a prevenir interferencia accidental entre las funciones. Si todas las variables fueran compartidas por el programa entero, sería muy difícil asegurarse de que algún nombre no fue usado para dos propósitos diferentes. Y si *reusaste* un nombre de variable, podrías ver efectos extraños de código no relacionado causando problemas en el valor de tu variable. la funcion al tratar tus variables locales, el lenguaje hace posible leer y entender las funciones como un pequeños universos, sin tener que preocuparse de todo el código a la vez.

Ámbitos Anidados

JavaScript distingue nos solo entre variables *globales* y *locales*. Las funciones pueden ser creadas dentro de otras funciones, produciendo distindos grados de localidad.

Por ejemplo, esta función sin sentido, tiene dos funciones dentro de ella:

```
var paisaje = function() {
  var resultado = "";
  var meseta = function(tamano) {
    for (var cuenta = 0; cuenta < size; cuenta++)
      resultado += "_";
  };
  var montana = function(tamano) {
    result += "/";
    for (var cuenta = 0; cuenta < size; cuenta++)
      resultado += "'";
    resultado += "\\";
  };

  meseta(3);
  montana(4);
  meseta(6);
  montana(1);
  meseta(1);
  return resultado;
};

console.log(landscape());
// → ___/''''\______/'\_
```

Las funciones `meseta` y `montana` pueden "ver" la variable llamada `resultado`, debido a que están dentro de la función que la define. Pero no pueden ver la variable `cuenta` entre ellas, porque están definidas fuera del ámbito de la otra. El entorno fuera de la función `paisaje` no puede acceder a ninguna de las variables definidas dentro de `paisaje`.

En pocas palabras, cada ámbito local también puede ver los ámbitos locales que lo contienen. El conjunto de variables visible dentro de la función es determinado por el lugar de la función en el texto del programa. Todas las variables de los bloques que *envuelven* una la definición de una función son visibles para ella, esto es, en todos los cuerpos de las funciones que la envuelven y las correspondientes al nivel superior(ámbito global). Este manera de resolver la visibilidad de las variables es llamada *definición léxica de ámbito*(lexical scoping).

Las personas que tienen experiencia con otros lenguajes de programción podrían esperar que cualquier bloque entre llaves produzca un nuevo entorno local. Pero en JavaScript, las funciones son lo único que crea un nuevo ámbito. Pero puedes usar bloques independientes.

```
var algo = 1;
{
  var algo = 2;
  // Haz algo con la variable...
}
// Fuera del bloque...
```

Pero la variable `algo` dentro del bloque se refiere a la misma variable que la que está fuera del bloque. De hecho, aunque bloques como este son permitidos, sólo son útiles para agrupar el cuerpo de las sentencias `if` o de los bucles.

Si esto te parece raro, no eres el único. La próxima versión de JavaScript introducirá una palabra reservada `let`, que trabaja como `var` pero crea una variable que es local para el *bloque* en el que está, no para la función.

Funciones como valores

Las variables de función normalmente actúan como nombres para una parte específica del programa. Esa variables es definida una vez y nunca es cambiada. Esto hace fácil empezar a confundir la función con su nombre.

Pero son diferentes entre sí. Un valor función puede hacer todo los que los otros valores pueden hacer; lo puedes usar en expresiones arbitrarias, no sólo llamarlos. Es posible guardar la función en un nuevo lugar, pasar como argumento a otra función, etc. De manera parecida, la variable que contiene una función sigue siendo una variable normal a la que se le puede asignar otro valor, como sigue:

```
var lanzarMisiles = function(valor) {
  sistemaDeMisiles.lanzar("ahora");
};
if (modoSeguro)
  lanzarMisiles = function(valor) {/* No hacer nada. */};
```

En Capítulo 5, hablaremos de las maravillosas cosas que puedes hacer al pasar valores función a otras funciones.

Notación de Declaración

Existe una forma más corta de decir *“var square = function…”*. La palabra reservada `function` puede ser usada al principio de una sentencia, como sigue:

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

Esta es una *declaración* de función. La expresión define la variable `cuadrado` y la apunta a la función dada. Hasta aquí todo bien. sin embargo, hay una pequeña sutileza con esta forma de definición.

```
console.log("El futuro dice: ", futuro());

function future() {
  return "Seguimos sin tener carros voladores.";
}
```

Este código funciona aunque la función está definida *debajo* del código que la usa. Esto es debido a que las declaraciones de función no toman parte en el flujo de control regular de arriba hacia abajo. Son movidas conceptualmente a la parte superior de su ámbito y pueden ser usadas por todo el código en ese ámbito. Esto es útil algunas veces por que nos da la libertad de organizar el código de una manera parezca significativa sin preocuparnos por definir todas las funciones antes de su primer uso.

¿Qué pasa cuando pones una declaración de función dentro de un bloque condicional (`if`) o dentro de un bucle?. Bueno, mejor no lo hagas. Diferentes plataformas de JavaScript en diferentes navegadores hacen diferentes cosas tradicionalmente en esa situación, y el último *estándar* de hecho lo prohibe. Si quieres que tus programas sean consistentes, usa las sentencias de declaración de función en el bloque más externo de tu función o programa.

```
function ejemplo() {
  function a() {} // Bien
  if (alguna_condicion) {
    function b() {} // ¡Peligro!
  }
}
```

La pila de llamadas

Será útil mirar más de cerca la forma en que el control se mueve a través de las funciones. Aquí hay un simple programa que hace unas cuantas llamadas a funciones.

```
function saluda(a_quien) {
  console.log("Hola " + a_quien);
}
saluda("Harry");
console.log("Adiós");
```

Una ejecución de este programa va más o menos así: la llamada a `saluda` causa que el control pase al inicio de esa función (línea 2). Esta llama a `control.log` (una función incluída en los navegadores), que toma el control, hace su trabajo, y devuelve el control a la línea 2. Después, se alcanza el final de la función `saluda`, así que se regresa al lugar en dónde se llamó, en la línea 4. La línea siguiente llama a `console.log` otra vez.

Podemos mostrar el flujo de control esquemáticamente así:

```
raíz
   saluda
        console.log
   saluda
raíz
   console.log
raíz
```

Debido a que una función tiene que saltar de regreso al lugar en que fue llamada cuando termine, la computadora debe recordar el contexto en el que fue llamada. En un caso, `console.log` tiene que regresar a la función `saluda`. En el otro caso salta al final del programa.

El lugar en el que la computadora guarda este contexto es la *pila de llamadas*. Cada vez que una función es llamada, el contexto actual es puesto en la parte superior de esta pila. Cuando la función retorne, remueve el contexto superior de la "pila" y lo usa para continuar la ejecución.

Guardar esta pila requiere espacio en la memoria de la coputadora. Cuando la pila se hace demasiado grande la computadora mostrará un error parecido a "out of stack space" (sin espacio en la en la pila) o "too much recursion" (demasiada recursión). El siguiente código lo ilustra al preguntarle algo realmente difícil a la computadora, lo que causa un ir y venir infinitamente entre funciones. Más bien, *sería* infinito, si la computadora tuviera una pila infinita. Como son las cosas, nos quedaremos sin espacio, o "volaremos la pila".

```
function gallina() {
  return huevo();
}
function huevo() {
  return gallina();
}
console.log(gallina() + " fue primero");
// → ??
```

Argumentos Opcionales

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

```
alert("Hola", "Buenas Noches", "¿Cómo estás?");
```

La función `alert` oficialmente acepta sólo un argumento. Aún así, cuando la llamas como aquí, no se queja. Simplemente ignora los otros argumentos y te muestra "Hola".

JavaScript es de mente extremadamente abierta sobre el número de argumentos que le pasas a una función. Sí le pasas demasiados, los argumentos extra son ignorados. Si le pasas muy pocos, los parámetros que faltan simplemente son asignados a `undefined`.

El lado malo de esto es que es posbible/probable, casi seguro que le pasarás accidentalmente un número incorrecto de argumentos a las funciones y nadie te avisará.

El lado bueno de este comportamiento es que puede ser usado para tener una función que tome parámetros "opcionales". Por ejemplo, la siguiente versión de `potencia` puede ser llamada con dos argumentos o uno solo, caso en el que el exponente se asume como dos, y la función se comporta como `cuadrado`.

```
function potencia(base, exponente) {
  if (exponente == undefined)
    exponente = 2;
  var resultado = 1;
  for (var cuenta = 0; cuenta < exponente; cuenta++)
    resultado *= base;
  return resultado;
}

console.log(potencia(4));
// → 16
console.log(potencia(4, 3));
// → 64
```

En el próximo capítulo, veremos una forma en la que el cuerpo de una función puede obtener la lista exacta de argumentos que se le pasaron. Esto es útil porque permite a una función aceptar un número indeterminado de argumentos. Por ejemplo, `console.log` hace uso de esto: imprime todos los valores que se le pasaron.

```
console.log("R", 2, "D", 2);
// → R 2 D 2
```

Closure

La habilidad de tratar *funciones* como valores, combinada con el hecho de que las varibles locales son "re-creadas" cada vez que una función es llamada, saca a la luz una pregunta interesante. ¿Qué pasa con las variables locales cuando la función que las creó ya no está activa?

El siguiente código muestra un ejemplo de esto. Define una funnción, `envuelveValor`, que crea una variable local. Después devuelve una función que accede y devuelve esta variable local.

```
function envuelveValor(n) {
  var variableLocal = n;
  return function() { return variableLocal; };
}

var envoltura1 = envuelveValor(1);
var envoltura2 = envuelveValor(2);
console.log(envoltura11());
// → 1
console.log(envoltura2());
// → 2
```

Esto está permitido y funciona como esperarías; la variable todavía puede leerse. De hecho, múltiples instancias de las variables pueden existir al mismo tiempo, lo que es otra buena ilustración del concepto de que las variables locales son re-creadas realmente para cada llamada; diferentes llamadas no pueden afectar otras variables locales.

Esta característica -ser capaces de hacer referencia a una instancia local de varables en un función que las encierra- se llama '*closure*'. Una función que "encapsula" algunas variables locales es llamada *un* closure. Este comportamiento no sólo te libera de preocuparte de los tiempos de vida de las variables, además permite algunos usos creativos de las funciones.

Con un pequeño cambio, podemos hacer del ejemplo anterior funciones que multipliquen por un número arbitrario.

```
function multiplicador(factor) {
  return function(numero) {
    return numero * factor;
  };
}

var doble = multiplicador(2);
console.log(doble(5));
// → 10
```

La variable explícita `variableLocal` de la función `envuelveValor` en el ejemplo previo no es necesaria porque un parámetro en sí mismo es una variable local.

Concebir los parámetros de esta forma requiere algo de práctica. Un buen modelo mental es pensar en la palabra clave `function` como si "congelara" el código que está dentro de ella en un paquete(el valor función). Así, cuando leas `return function(...){...}`, piensa en que esto regresa un acceso a un conjunto de cálculos, congelados para uso posterior.

En el ejemplo, `multiplicador` regresa un pedazo congelado de código que se guarda en la variable `doble`. La última línea entonces llama el valor guardado en esta variable, haciendo que el código congelado (`return numero * factor;`) se active. Este todavía tiene acceso a la variable `factor` de la llamada a `multiplicador` que lo creó, y además obtiene acceso al argumento que se pasa cuando se activa el código, `5`, a través de su parámetro `numero`.

Recursión

Es perfectamente correcto que una función se llame a sí misma, mientras tenga cuidado de no desbordar la pila. Una función que se llama a sí misma se llama *recursiva*. La recursión permite que algunas funciones se escriban con un estilo diferente. Tomemos por ejemplo, esta implementación alternativa de `potencia`:

```
function potencia(base, exponente) {
  if (exponente == 0)
    return 1;
  else
    return base * potencia(base, exponente - 1);
}

console.log(potencia(2, 3));
// → 8
```

Esto es más cercano al modo en que los matemáticos definen la potenciación y describe el concepto de un modo más elegante que la variante que lo hace con bucles. La función se llama a sí misma varias veces con diferentes argumentos para conseguir la multiplicación

repetida.

Pero esta implementación tiene un problema importante: en implementaciones típicas de JavaScript, es cerca de 10 veces más lenta que la versión con bucles. Correr a través de un siple bucle es más barato que llamar a una función muchas veces.

El dilema de velocidad contra *elegancia* es interesante. Puedes verlo como una lucha continua entre amigabilidad-humano y amigabilidad-máquina. Casi cualquier programa puede hacerse más rápido haciéndolo más grande y convolucionado. El programador debe de decidir el balance apropiado.

En el caso de la función anterior `potencia` la poco elegante versión(iterativa) es aún bastante simple y fácil de leer. No tiene mucho sentido reemplazarla con la versión recursiva. A menudo, sin embargo, un programa tiene conceptos tan complejos que sacrificar un poco de eficiencia para hacer el programa más claro se vuelve una opción atractiva.

La regla básica, que ha sido repetida por muchos programadores y con la cuál concuerdo de todo corazón, es no preocuparse por la eficiencia hasta que estés seguro que el programa es demasiado lento. Si lo es, busca las partes que están abarcando la mayoría del tiempo y empieza a cambiar *elegancia* por *eficiencia* en esas partes.

Por supuesto, esta regla no significa que debas ignorar el rendimiento completamente. En muchos casos, como en la función `potencia`, no se gana demasiada simplicidad de la solución "elegante". Y a veces un programador experimentado puede ver de inmediato que un enfoque sencillo nunca va a ser lo suficientemente rápido.

La razón por la que estoy hablando tanto de esto es que muchos programadores novatos se enfocan fanáticamente en la eficiencia, incluso en los detalles más pequeños. El resultado son programas más grandes, más complicados y a menudo menos correctos, que toman más tiempo en escribirse que sus equivalentes más sencillos y que generalmente corren solo un poco más rápido.

Pero la recursión no es siempre sólo una alternativa menos eficiente a los bucles. Alguos problemas son mucho más fáciles de resolver con recursión que con bucles. La mayoría de estos son problemas que requieren explorar o procesar varias "ramas", cada una de las cuáles se puede ramificar otra vez.

Considera el siguente rompecabezas: empezando por el número 1 y añadiendo repetidamente 5 o multiplicándolo por 3, un número infinito de nuevos números puede ser producido. ¿Cómo escribirías una función que, dado un número, trate de encontrar una secuencia de sumas y multipliclaciones que producen ese número? Por ejemplo, el número 13 puede ser producido al multiplicar por 3 primero y después sumar 5 dos veces, mientras que el número 15 no puede ser producido.

Aquí hay una solución recursiva:

```
function encontrarSolucion(objetivo) {
  function encontrar(inicio, historia) {
    if (inicio == objetivo)
      return historia;
    else if (inicio > objetivo)
      return null;
    else
      return encontrar(inicio + 5, "(" + historia + " + 5)") ||
             encontrar(inicio * 3, "(" + historia + " * 3)");
  }
  return encontrar(1, "1");
}

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

Nota que este programa no encuentra necesariamente la ruta *más corta* de operaciones. Se satisface cuando cuentre cualquier sequencia.

No necesariamente espero que veas como este trabaja esto inmediatamente. Pero trabajemos en eso, porque es un gran ejercicio en el pensamiento recursivo.

La función interna `encontrar` hace la recursividad. Toma dos argumentos – el número actual y una cadena que registra como hemos alcanzado el número – y regresa una cadena que muestra como llegar al objetivo o `null`.

Para hacer esto la función realiza una de tres acciones. Si el número actual es el número objetivo, entonces la historia actual es una forma de alcanzar este objetivo, así que siemplemente es devuelta. Si el número actual es mayor que el número objetivo, no tiene sentido seguir haciendo más exploraciones porque tanto sumar como multiplicar sólo hará mayor el número. Y finalmente, si estamos todavía debajo del objetivo la función prueba los dos caminos posibles que empiezan con el número actual llamándose dos veces a sí misma, una vez por cada uno de los pasos permitos. Si la primera llamada regresa cualquier cosa que no sea `null`, se devuelve como resultado. De otra forma, la segunda opción es la que se devuelve, independientemente de si produce una cadena o `null`.

Para entender mejor como esta función produce el efecto que estamos buscando, miremos a todas las llamadas a `encontrar` que se hacen cuando estamos buscando la solución para el número 13.

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

El sangrado (indentación) indica la profundidad en la pila de llamadas. La primera vez que `encuentra` es llamada, se llama dos veces a sí misma para explorar las soluciones que empiezan con `(1 + 5)` y `(1 * 3)`. La primera llamada intenta encontrar una solución que empiece con `(1 + 5)` y, usando la recursión, explora *todas* las soluciónes que produzcan un número menor o igual que el número buscado. Dado que no encuentra una solución que corresponda con el objetivo, regresa `null` a la primera llamada. Entonces el operador `||` hace que la llamada que explora `(1 * 3)` suceda. Esta búsqueda tiene más suerte debido a que su primera llamada recursiva, a través de *otra* llamada recursiva, llega al número que buscamos, 13. Esta llamada recursuva, la más interna, regresa una cadena y cada uno de los operadores `||` en la llamada superior inmediata pasan esa cadena, finalmente regresando nuestra solución.

Funciones crecientes

Existen dos formas más o menos naturales de intorducir las funciones en los programas.

La primera es que te encuentras a ti mismo escribiendo el mismo código varias veces. Necesitamos evitar hacer eso, debido a que más código significa más espacio para que los errores se oculten y más material para que leer para las personas que quiere entender el programa. Así que tomamos funcionalidad repetida, encontramos un buen nombre para esta, y la ponemos dentro de una función. La segunda forma es que encuentres que necesitas cierta funcionalidad que no has escrito aún y que suena como a que merece su propia función. Empezarás por nombrar la función y después escribirás su cuerpo. Podrías incluso empezar a escribir el código que usa la función antes de realmente definir la función misma.

Qué tan difícil es encontrar un nombre para una función, es un buen indicador de que tan claro tienes el concepto de aquello que estás tratando de envolver. Hagamos un ejemplo.

Necesitamos escribir un problema que imprima dos números, el número de las vacas y de los pollos de una granja, con las palabras `Vacas` y `Pollos` después de estos, y completados con ceros para que siempre sean de 3 dígitos de longitud.

```
007 Vacas
011 Pollos
```

Esto claramente requiere una función de dos argumentos. Empecemos a programar.

```
function imprimeInventarioGranja(vacas, pollos) {
  var vacasString = String(vacas);
  while (vacasString.length < 3)
    vacasString = "0" + vacasString;
  console.log(vacasString + " Vacas");
  var pollosString = String(pollos);
  while (pollosString.length < 3)
    pollosString = "0" + pollosString;
  console.log(pollosString + " Pollos");
}
imprimeInventarioGranja(7, 11);
```

Añadir `.length` despues de un valor de cadena nos dará la longitud de esa cadena. Así, los `while` continuán agregando ceros al principio de la cadena del número hasta que son por lo menos de 3 caracteres.

¡Misión cumplida! Pero casi cuando le vamos a mandar el código al granjero (con una buena factura) nos llama y nos dice que ha empezado a críar puercos, y que si podríamos extender el software para también mostrar los puerquitos, ¿por favor?.

De seguro podemos. Pero mientras estamos en el proceso de copiar y pegar esas cuatro líneas una vez más, paramos y reconsideramos. Existe una mejor forma. Aquí hay un intento:

```
function imprimeConCerosYEtiqueta(numero, etiqueta) {
  var numeroString = String(numero);
  while (numeroString.length < 3)
    numeroString = "0" + numeroString;
  console.log(numeroString + " " + etiqueta);
}

function imprimeInventarioGranja(vacas, pollos, puercos) {
  imprimeConCerosYEtiqueta(vacas, "Vacas");
  imprimeConCerosYEtiqueta(pollos, "Pollos");
  imprimeConCerosYEtiqueta(puercos, "Puercos");
}

imprimeInventarioGranja(7, 11, 3);
```

¡Funciona! Pero ese nombre, `imprimeConCerosYEtiqueta`, es un poco feo. Amontona tres cosas–imprimir, completar con ceros, y agragar la etiqueta–en una sola función.

En vez de quitar la parte repetida de nuestra programa por mayoreo, tratemos de escoger un solo *concepto*.

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

function imprimeInvantarioGranja(vacas, pollos, puercos) {
  console.log(rellenoConCero(vacas, 3) + " Vacas");
  console.log(rellenoConCero(pollos, 3) + " Pollos");
  console.log(rellenoConCero(puercos, 3) + " Puercos");
}

imprimeInvantarioGranja(7, 16, 3);
```

Una función con un bonito y obvio nombre como `rellenoConCero` hace más fácil para alguien que lea el código descubrir lo que hace. Y eso es útil en más situaciones que sólo en este programa específico. Por ejemplo, puedes usarla para imprimir una tabla bien alineada de números.

¿Qué tan inteligente y versátil debería ser nuestra función? Podríamos escribir cualquier cosa desde una función terriblemente simple que sólamente rellena un número de tal manera que tenga 3 caracteres hasta una complicada, muy generalizada función, un sistema de formateo de números que maneje fracciones, números negativos, alineación de puntos, relleno con diferentes carecteres y así por el estilo.

Un principio útil, no añadir características a menos que estés completamente seguro de que las vas a ocupar. Puede ser tentador escribir marcos de trabajo generals "_framework_s" para cada pequeño pedazo de funcionalidad que te encuentras. Resiste el impulso. No acabarás ningún trabajo real y terminarás escribiendo mucho código que nadie usará alguna vez.

Funciones y efectos colaterales

Las funciones pueden ser más o menos divididas en aquellas que son llamadas por sus efectos colaterales y aquellas que son llamadas por su valor de resultado. (Aunque es completamente posible tener tanto efectos colaterales como retornar un valor).

La primer función de ayuda en el ejemplo de la granja, `imprimeConCerosYEtiqueta`, es llamada por su efecto colateral: imprime su valor de resultado. La segunda versión, `rellenoConCero`, es llamada por su valor de resultado. No es coincidencia que la segunda sea útil en más situaciones que la primera. Las funciones que crean valores son más fáciles de combinar en nuevas formas que funciones que realizan directamente un efecto colateral.

Una función *pura* es un tipo de función que produce un valor que no sólo carece de efectos colaterales, tampoco usa los efectos colaterales de otro código–por ejemplo, no lee variables globales que son ocasionalmente cambiadas por otro código. Una función pura tiene la agradable propiedad de que, cuando se llama a los mismos argumentos, siempre produce el mismo valor (y no hace nada más). Esto hace fácil razonar con ella. Una llamada a una función como esta puede ser sustituida mentalmente como su resultado, sin cambiar el significado del código. Cuando no estas seguro de que una función pura funciona correctamente, puedes probarlo simplemente llamándola, y sabiendo que trabaja bien este contexto, trabajará bien en cualquier contexto. Funciones no puras pueden regresar diferentes valores basadas en todo tipo de factores y tienen efectos secundarios y que pueden ser difícil de probar y de razonar con ellas.

Aún así, no hay necesidad de sentirse mal cuando escribimos funciones que no son puras o de armar una guerra santa para eliminarlas de tu código. Los efectos secundarios a menudo son útiles. No habría forma de imprimir una versión pura de `console.log`, por ejemplo, y `console.log` es ciertamente útil. Algunas operaciones son además más fáciles de expresar en una forma eficiente cuando se usan los efectos secundarios, así que la velocidad de cómputo puede ser una razón para evitar la pureza.

Resumen

Este capítulo se enseñó como escribir tus propias funciones. La palabra resevada `function`, cuando se usa como una expresión, puede crear un valor función. Cuando es usada como sentencia, puede ser usada para declarar una variable y darle una función

como valor.

```
// Crear un valor función f
var f = function(a) {
  console.log(a + 2);
};

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

Un aspecto clave para entender las funciones es entender los ámbitos locales. Los parámetros y variables declaradas dentro de una función son locales para la función, re-creados cada vez que la función es llamada y no visibles de desde fuera. Las funciones declaradas dentro de otra función tienen acceso al ámbito local externo de la función.

Separar las tareas que tu tarea realiza en diferentes funciones es útil. No tendrás que repetir lo mismo tanto, y las funciones pueden hacer un programa más legible al agrupar el código en partes conceptuales, de la misma forma que capítulos y secciones ayudan a organizar el texto regular.

Ejercicios

Mínimo

El capítulo anterior intordujo la función estándar `Math.min` que devuelve su argumento más pequeño. Ahora nosotros mismos podemos hacer eso. Escribe una función `min` que tome dos argumentos y devuelva el mínimo.

```
// Tu código va aquí.

console.log(min(0, 10));
// → 0
console.log(min(0, -10));
// → -10
```

!!pista!!

Si tienes problemas poniendo las llaves y los paréntesis en el lugar correcto para obtener una definición de función válida, empieza por copiar uno de los ejemplos del capítulo y modificarlo.

Una función puede contener múltiples `return`.

Recursión

Hemos visto que `%` (el operador de sobrante) puede ser usado para probar si un número es par o impar, usando `% 2` para checar si es divisible por dos. Aquí hay otra forma de definir si un número entero es par o impar:

- Cero es par.
- Uno es impar.
- Para cualquier otro número *N*, su paridad es la misma que *N* - 2.

Escribe una función recursiva `esPar` que corresponda a esta descripción. La función debería aceptar un `numero` como parámetro y regresar un Booleano.

Pruebala con 50 y 75. Observa cómo se comporta con -1. ¿Por qué? ¿Puedes pensar en alguna forma de arreglar esto?

```
// Tu código va aquí.

console.log(esPar(50));
// → true
console.log(esPar(75));
// → false
console.log(esPar(-1));
// → ??
```

!!pista!!

La función será algo similar al `encuentra` interno en la solución recursiva `encuentraSolucion` en este capítulo, con una cadena de `if` / `else if` / `else` que probará cuál de los tres casos aplica. El `else` final, correspondiente al tercer caso, hace la llamada recursiva. Cada una de las ramas debe de contener una sentencia `return` o de alguna otra forma arreglárselas para devolver un valor específico.

Cuando se le da un número negativo, la función va a llamarse a sí misma una y otra vez, pasándose un número cada vez más negativo, así alejándose más y más de regresar una solución. En algún momento se quedará sin espacio en la pila y abortará.

Contando Frijoles

Puedes obtener el n-avo caracter, o letra, de una cadena escribiendo `"cadena".charAt(N)`, similar a como obtienes su longitud con `"cadena".length`. El valor obtenido será una cadena que contiene sólo un caracter(por ejemplo, `"b"`). El primer caracter tiene la

posición cero, lo cual hace que el último pueda ser encontrado en la posición `string.lenght -1` . En otras palabras, una cadenas de dos caracteres tiene un “lenght” o longitud de 2, y sus caracteres tienen las posiciones 0 y 1.

Escribe una función `cuentaFs` que tome una cadena como su único argumento y regrese un número que indique cunántos caracteres “F” mayúscula hay en la cadena.

A continuación, escribe una funcióon llamada `cuentaCaracter` que se comoporte como `cuentaFs` , con la diferencia de que tome un segundo caracter que indique el caracter que será contado(en vez de sólo caracteres “F”). Reescribe `cuentaFs` para hacer uso de esta nueva función.

```
// Tu código va a aquí.

console.log(cuentaFs("FFC"));
// → 2
console.log(cuentaCaracter("ferrocarril", "r"));
// → 4
```

!!pista!!

Un bucle en tu función tendrá que revisar cada caracter en la cadena corriendo un índice desde cero hasta uno menos que su longitud (`< cadena.length`). Si el caracter de la posición actual es el mismo que la función está buscando, añade uno a la variable contador. Una vez que se ha terminado el bucle, el contador puede ser regresado.

Pon atención en hacer todas las variables usadas en la función *locales* a la función midiante el uso de la palabra reservada `var` .

Estructuras de Datos: Objetos y Arreglos

> En dos ocasiones me preguntaron - “Disculpe, Sr. Babbage, si pongo números incorrectos en la máquina, ¿van a salir las respuestas correctas?” [...] No puedo terminar de comprender el tipo de confusión de ideas que podrían provocar esta pregunta. Charles Babbage, Passages from the Life of a Philosopher (1864)

Números, Booleanos y cadenas son los ladrillos de los que están hechas las estructuras de datos. Pero no podrás construir mucha casa de un solo ladrillo. Los *objetos* nos permiten agrupar valores-incluyendo otros objetos-permitiéndonos construir estructuras más complejas.

Los programas que hemos construido hasta ahora han sido seriamente limitados debido al hecho de que estaban operando únicamente en tipos de datos simples. Este capítulo agregará a tu caja de herramientas un entendimiento básico de las estructuras de datos. Al finalizarlo, sabrás lo suficiente para empezar a escribir algunos programas de utilidad.

El capítulo trabajará a lo largo de un ejemplo de programación más o menos realista, introduciendo conceptos conforme apliquen al problema en cuestión. El código de ejemplo muchas veces se construirá sobre funciones y variables que fueron presentadas previamente en el texto.

El sandbox de programación en línea para el libro *eloquentjavascript.net/code* proporciona una manera de correr el código en el contexto de un capítulo en espacífico. Si decides trabajar en los ejemplos en otro entorno, asegúrate de descargar primero el código completo de este capítulo desde la página del sandbox.

La ardillalobo

De vez en cuando, comúnmente entre las ocho y las diez de la noche, Jacques se transforma en un pequeño y peludo roedor con una frondosa cola.

Por un lado, Jacques esta bastante contento de no tener la clásica licantropía. Convertirse en una ardilla suele causar menos problemas que convertirse en un lobo. En vez de tener que preocuparse por comerse accidentalmente a un vecino (*eso* sería extraño), le preocupa el ser deborado por el gato del vecino. Después de un par de ocasiones donde se despertó, desnudo y desorientado, en una apenas delgada rama en la cima de un roble, se ha asegurado de cerrar puertas y ventanas de su cuarto por las noches y poner algunas nueces en el suelo para mantenerse ocupado.

Eso resuelve los problemas del gato y el roble. Pero Jacques aún sufre de su enfermedad. Los momentos irregulares en que se presenta la transformación le hacen sospechar que pudieran ser detonadas por algo. Por algún tiempo, creyó que sucedía sólo en los dias que había tocado árboles. Así que dejó de tocar árboles de manera definitiva e incluso evitó acercarse a ellos. Pero el problema presistió.

Cambiando a una perspectiva un poco más científica, Jacques planea empezar un registro diario de todo lo que hizo ese día y si tuvo o no una transformación. Con estos datos espera limitar las condiciones que disparan las transformaciones.

La primer cosa que hace es diseñar una estructura de datos para almacenar esta información.

Conjuntos de datos

Para trabajar con un pedazo de datos digitales, primero tendremos que encontrar una forma de representarlo en la memoria de nuestra máquina. Digamos, como un pequeño ejemplo, que queremos representar una colección de números: 2, 3, 5, 7 y 11.

Podríamos ponernos creativos usando cadenas-despues de todo, las cadenas pueden ser de cualquier longitud, así que podríamos pone mucha información en ellas-y usar `"2 3 5 7 11"` como nuestra representación. Pero esto es extraño. De alguna forma tendrías que extaer los dígitos y convertirlos de vuelta a números para accesarlos.

Afortunadamente, Javascript proporciona un tipo de dato específico para almacenar secuencias de valores. Se le llama *arreglo* y se escribe como una lista de valores entre corchetes, separados por comas.

```
var listOfNumbers = [2, 3, 5, 7, 11];
console.log(listOfNumbers[1]);
// → 3
console.log(listOfNumbers[1 - 1]);
// → 2
```

La notación para obtener los elementos dentro de un arreglo también utiliza corchetes. Un par de corchetes inmediátamente después de una expresión, con otra expresión dentro de los corchetes. buscará el elemento en la expresión de la izquierda que corresponda al *índice* dado por la expresión en corchetes.

El primer índice de un arreglo es cero, no uno. Así que el primer elemento puede leerse usando `listOfNumbers[0]`. Si no tienes antecedentes en programación, acostumbrarte a esta convención puede tomarte algún tiempo. Pero el *zero-based counting* tiene una larga tradición en tecnología y mientras la convención se siga de manera consistente (que se ha hecho, en Javascript), funciona bien.

Propiedades

Hemos visto algunas expresiones sospechosas como `myString.length` (para obtener la longitud de una cadena) y `Math.max` (la función máximo) en ejemplos pasados. Estas son expresiones que accesan una *propiedad* de algún valor. En el primer caso, accesamos la propiedad `length` de el valor en `myString`. En el segundo, accesamos la propiedad llamada `max` en el objeto `Math` (que es una colección de valores y funciones relacionadas con las matemáticas).

Casi todos los valores de Javascript tienen propiedades. Las excepciones son `null` y `undefined`. Si intentas acceder una propiedad de alguno de estos nonvalues, recibirás un error.

```
null.length;
// → TypeError: Cannot read property 'length' of null
```

Las dos maneras comúnes de acceder a propiedades en Javascript es con un punto y con corchetes. Ambas `value.x` y `value[x]` acceden una propiedad en *value*—pero no necesariamente la misma propiedad. La diferencia radica en cómo se interpreta `x`. Cuando usamos un punto, la parte después del punto debe ser un nombre de variable válido y nombra de manera directa a la propiedad. Cuando usamos corchetes, la expresión dentro de los corchetes es *evaluada* para obtener el nombre de la propiedad. Mientras que `value.x` busca la propiedad de `value` llamada “x”, `value[x]` intenta evaluar la expresión `x` y usa el resultado como el nombre de la propiedad.

Así que si sabes que la propiedad que te interesa se llama “length”, usas `value.length`. Si deseas extraer la propiedad nombrada por el valor almacenado en la variable `i`, usas `value[i]`. Y debido a que el nombre de las propiedades puede ser cualquier cadena, si quieres accesar una propiedad llamada “2” o “John Doe”, debes utilizar corchetes: `value[2]`

or `value["John Doe"]`. Así lo harías incluso si conoces el nombre preciso de la propiedad de antemano, ya que ni “2” ni “John Doe” son nombres válidos de variables y por lo tanto no pueden accesarse a traves de la notación con punto.

Los elementos en un arreglo se almacenan en propiedades. Debido a que los nombres de estas propiedades son números y usualmente necesitamos obtener su nombre de una variable, tenemos que usar la sintaxis de corchetes para accesarlos. La propiedad `length` de un arreglo nos dice cuantos elementos contiene. Este nombre de propiedad es un nombre de variable válido, y conocemos su nombre por anticipado, así que para encontrar la longitud de un arreglo, comúnmente escribiremos `array.length` ya que es más fácil de escribir que `array["length"]`.

Métodos

Ambos objetos, las cadenas y los arreglos contienen, adicionalmente a la propiedad `length`, un número de propiedades que refieren a valores de tipo función.

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

Todas las cadenas tienen una propiedad `toUpperCase`. Cuando es llamada, regresará una copia de la cadena en la que todas las letras han sido convertidas a mayúsculas. También existe `toLowerCase`. Puedes adivinar que es lo que hace.

Curiosamente, a pesar de que la llamada a `toUpperCase` no pasa ningun argumento, la funcion del algun modo tiene acceso a la cadena `"Doh"`, el valor cuya propiedad hemos llamado. Como funciona esto es decrito en el Capitulo 6.

Las propiedades que contienen funciones son generalmente llamadas metodos del valor al que pertenecen. Como , *“toUpperCase* es un metodo de una cadena”.

Este ejemplo demuestra algunos de los métodos que los objetos de tipo array tienen:

```
var mack = [];
mack.push("Mack");
mack.push("the", "Knife");
console.log(mack);
// → ["Mack", "the", "Knife"]
console.log(mack.join(" "));
// → Mack the Knife
console.log(mack.pop());
// → Knife
console.log(mack);
// → ["Mack", "the"]
```

El metodo `push` puede ser usado para añadir valores al final de un arreglo. El metodo pop hace lo opuesto: remueve el valor al final del arreglo y lo retorna. Un arreglo de cadenas puede ser aplanado a una sola cadena con el metodo join. El argumento dado a join determina el texto que es pegado entre los elementos del arreglo.

Objetos

De regreso a la ardillalobo. Un conjunto de registro de entradas puede ser representado como un arreglo. Pero las entradas no consisten solamente en un numero o una cadena- cada entrada necesita almacenar una lista de actividades y un valor Booleano que indique si Jacques se convirtió en una ardilla. Idealmente, nos gustaría agrupar esos valores juntos en un unico valor y despues poner esos valores agrupados en un arreglo de registro de entradas.

Los valores del tipo *object* son colecciones arbitrarias de propiedades y podemos agregar o eliminar esas propiedades como nos parezca. Una manera de crear un objeto es usar notación de llaves.

```
var day1 = {
  squirrel: false,
  events: ["work", "touched tree", "pizza", "running", "television"]
};
console.log(day1.squirrel);
// → false
console.log(day1.wolf);
// → undefined
day1.wolf = false;
console.log(day1.wolf);
// → false
```

Dentro de las llaves, podemos dar una lista de propiedades separadas por comas. Cada propiedad está escrita como un nombre, seguida por dos puntos, seguida por una expresión que provee un valor para la propiedad. Los espacios y saltos de línea no son significantes.

Cuando un objeto abarca multiples líneas, marcandolos como en el ejemplo anterior, esto mejora la legibilidad del codigo. Las propiedades cuyos nombres no son nombres validos de variables o numeros validos tienen que ser encerradas en comillas.

```
var descriptions = {
  work: "Went to work",
  "touched tree": "Touched a tree"
};
```

Esto significa que las *llaves* tienen *dos* significados en JavaScript. Al inicio de una sentencia, inician un bloque de sentencias. En cualquier otra posición, describen un objeto. Afortunadamente, casí nunca es util iniciar una declaración con un objeto de tipo *llave*, y en programas típicos, no hay ambigüedad entre estos dos usos.

Leer una propiedad que no existe producirá el valor undefined, lo que paso la primera vez que intentamos leer la propiedad lobo en el ejemplo anterior.

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

Para volver brevemente a nuestro modelo de tentáculos de asociacion de *variables*. Asociación de variables similares. Ellos captan valores, pero otras variables y propiedades podrían estar llevandose a cabo en los mismos valores. Tú puedes pensar en objetos como los pulpos con cualquier número de tentáculos, cada uno de los cuales tiene un nombre inscrito en ella.

Es un operador unitario que, cuando se aplica a una expresión acceso a la propiedad, eliminará la propiedad con el nombre del objeto. Esto no es una cosa común a hacer, pero es posible.

```
var anObject = {left: 1, right: 2};
console.log(anObject.left);
// → 1
delete anObject.left;
console.log(anObject.left);
// → undefined
console.log("left" in anObject);
// → false
console.log("right" in anObject);
// → true
```

El operador binario 'in', cuando se aplica a una cadena y un objeto, devuelve un valor booleano que indica si ese objeto tiene esa propiedad. La diferencia entre el establecimiento de una propiedad a `undefined`, y el hecho de eliminarlo es que, en el

primer caso, el objeto todavía tiene la propiedad (que simplemente no tiene un valor muy interesante), mientras que en el segundo caso la propiedad ya no está presente y devolverá falso.

Arrays, then, are just a kind of object specialized for storing sequences of things. If you evaluate `typeof [1, 2]`, this produces `"object"`. You can see them as long, flat octopuses with all their arms in a neat row, labeled with numbers.

image::img/octopus-array.jpg[alt="Artist's representation of an array"]

So we can represent Jacques' journal as an array of objects.

```
var journal = [
  {events: ["work", "touched tree", "pizza",
            "running", "television"],
   squirrel: false},
  {events: ["work", "ice cream", "cauliflower",
            "lasagna", "touched tree", "brushed teeth"],
   squirrel: false},
  {events: ["weekend", "cycling", "break",
            "peanuts", "beer"],
   squirrel: true},
  /* and so on... */
];
```

== Mutability ==

We will get to actual programming *real* soon now. But first, there's one last piece of theory to understand.

We've seen that object values can be modified. The types of values discussed in earlier chapters, such as numbers, strings, and Booleans, are all **immutable**—it is impossible to change an existing value of those types. You can combine them and derive new values from them, but when you take a specific string value, that value will always remain the same. The text inside it cannot be changed. If you have reference to a string that contains `"cat"`, it is not possible for other code to change a character in *that* string to make it spell `"rat"`.

With objects, on the other hand, the content of a value *can* be modified by changing its properties.

When we have two numbers, 120 and 120, we can consider them precisely the same number, whether or not they refer to the same physical bits. But with objects, there is a difference between having two references to the same object and having two different objects that contain the same properties. Consider the following code:

```
var object1 = {value: 10};
var object2 = object1;
var 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
```

(((tentacle (analogy))))(((variable,model of)))The `object1` and `object2` variables grasp the *same* object, which is why changing `object1` also changes the value of `object2`. The variable `object3` points to a different object, which initially contains the same properties as `object1` but lives a separate life.

(((== operator)))(((comparison,of objects)))(((deep comparison)))JavaScript's `==` operator, when comparing objects, will return `true` only if both objects are precisely the same value. Comparing different objects will return `false`, even if they have identical contents. There is no “deep” comparison operation built into JavaScript, which looks at object's contents, but it is possible to write it yourself (which will be one of the link:04_data.html#exercise_deep_compare[exercises] at the end of this chapter).

== The lycanthrope's log ==

(((weresquirrel example)))(((lycanthropy)))(((addEntry function)))So Jacques starts up his JavaScript interpreter and sets up the environment he needs to keep his ((journal)).

// include_code

```
var journal = [];

function addEntry(events, didITurnIntoASquirrel) {
  journal.push({
    events: events,
    squirrel: didITurnIntoASquirrel
  });
}
```

And then, every evening at ten—or sometimes the next morning, after climbing down from the top shelf of his bookcase—he records the day.

```
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);
```

Once he has enough data points, he intends to compute the ((correlation)) between his squirrelification and each of the day's events and ideally learn something useful from those correlations.

(((correlation)))*Correlation* is a measure of ((dependence)) between ((variable))s ("variables" in the statistical sense, not the JavaScript sense). It is usually expressed as a coefficient that ranges from -1 to 1. Zero correlation means the variables are not related, whereas a correlation of one indicates that the two are perfectly related—if you know one, you also know the other. Negative one also means that the variables are perfectly related but that they are opposites—when one is true, the other is false.

(((phi coefficient)))For binary (Boolean) variables, the *phi* coefficient (ϕ) provides a good measure of correlation and is relatively easy to compute. To compute ϕ, we need a ((table)) *n* that contains the number of times the various combinations of the two variables were observed. For example, we could take the event of eating ((pizza)) and put that in a table like this:

image::img/pizza-squirrel.svg[alt="Eating pizza versus turning into a squirrel",width="7cm"]

ϕ can be computed using the following formula, where *n* refers to the table:

ifdef::html_target[]

++++

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

 ++++

endif::html_target[]

ifdef::tex_target[]

pass:[\begin{equation}\varphi = \frac{n*{11}n*{00}-n*{10}n*{01}}
{\sqrt{n*{1\bullet}n*{0\bullet}n*{\bullet1}n*{\bullet0}}}\end{equation}]

endif::tex_target[]

The notation (!html *n*~01~!)(!tex pass:[$n*{01}$]!) *indicates the number of measurements where the first variable (squirrelness) is false (0) and the second variable (pizza) is true (1). In this example, (!html _n*~01~!)(!tex pass:[n_{01}]!) is 9.

The value (!html *n*~1•~!)(!tex pass:[$n*{1\bullet}$]!) *refers to the sum of all measurements where the first variable is true, which is 5 in the example table. Likewise, (!html _n*~•0~!)(!tex pass:[$n_{\bullet0}$]!) refers to the sum of the measurements where the second variable is false.

(((correlation)))(((phi coefficient)))So for the pizza table, the part above the division line (the dividend) would be 1×76 - 4×9 = 40, and the part below it (the divisor) would be the square root of 5×85×10×80, or (!html √340000!)(!tex pass:[$\sqrt{340000}$]!). This comes out to ϕ ≈ 0.069, which is tiny. Eating ((pizza)) does not appear to have influence on the transformations.

== Computing correlation ==

(((array,as table)))(((nesting,of arrays)))We can represent a two-by-two ((table)) in JavaScript with a four-element array (`[76, 9, 4, 1]`). We could also use other representations, such as an array containing two two-element arrays (`[[76, 9], [4, 1]]`) or an object with property names like `"11"` and `"01"` , but the flat array is simple and makes the expressions that access the table pleasantly short. We'll interpret the indices to the array as two-((bit)) ((binary number)), where the leftmost (most significant) digit refers to the squirrel variable and the rightmost (least significant) digit refers to the event variable. For example, the binary number `10` refers to the case where Jacques did turn into a squirrel, but the event (say, "pizza") didn't occur. This happened four times. And since binary `10` is 2 in decimal notation, we will store this number at index 2 of the array.

(((phi coefficient)))(((phi function)))This is the function that computes the ϕ coefficient from such an array:

// test: clip // include_code strip_log

```
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
```

(((square root)))(((Math.sqrt function)))This is simply a direct translation of the ϕ formula into JavaScript. `Math.sqrt` is the square root function, as provided by the `Math` object in a standard JavaScript environment. We have to sum two fields from the table to get fields like (!html n~1•~!)(!tex pass:[$n_{1\bullet}$]!) because the sums of rows or columns are not stored directly in our data structure.

(((JOURNAL data set)))Jacques kept his journal for three months. The resulting ((data set)) is available in the coding sandbox for this chapter(!book (http://eloquentjavascript.net/code#4[_eloquentjavascript.net/code#4_])!), where it is stored in the `JOURNAL` variable, and in a downloadable http://eloquentjavascript.net/code/jacques_journal.js[file].

(((tableFor function)))(((hasEvent function)))To extract a two-by-two ((table)) for a specific event from this journal, we must loop over all the entries and tally up how many times the event occurs in relation to squirrel transformations.

// include_code strip_log

```
function hasEvent(event, entry) {
  return entry.events.indexOf(event) != -1;
}

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

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

(((array,searching)))(((indexOf method)))The `hasEvent` function tests whether an entry contains a given event. Arrays have an `indexOf` method that tries to find a given value (in this case, the event name) in the array and returns the index at which it was found or -1 if it wasn't found. So if the call to `indexOf` doesn't return -1, then we know the event was found in the entry.

(((array,indexing)))The body of the loop in `tableFor` figures out which box in the table each journal entry falls into by checking whether the entry contains the specific event it's interested in and whether the event happens alongside a squirrel incident. The loop then adds one to the number in the array that corresponds to this box on the table.

We now have the tools we need to compute individual ((correlation))s. The only step remaining is to find a correlation for every type of event that was recorded and see whether anything stands out. But how should we store these correlations once we compute them?

== Objects as maps ==

(((weresquirrel example)))(((array)))One possible way is to store all the ((correlation))s in an array, using objects with `name` and `value` properties. But that makes looking up the correlation for a given event somewhat cumbersome: you'd have to loop over the whole array to find the object with the right `name`. We could wrap this lookup process in a function, but we would still be writing more code, and the computer would be doing more work than necessary.

[[object_map]] (((object)))(((square brackets)))(((object,as map)))(((in operator)))A better way is to use object properties named after the event types. We can use the square bracket access notation to create and read the properties and can use the `in` operator to test whether a given property exists.

```
var map = {};
function storePhi(event, phi) {
  map[event] = phi;
}

storePhi("pizza", 0.069);
storePhi("touched tree", -0.081);
console.log("pizza" in map);
// → true
console.log(map["touched tree"]);
// → -0.081
```

(((data structure)))A *((map))* is a way to go from values in one domain (in this case, event names) to corresponding values in another domain (in this case, ϕ coefficients).

There are a few potential problems with using objects like this, which we will discuss in link:06_object.html#prototypes[Chapter 6], but for the time being, we won't worry about those.

(((for/in loop)))(((for loop)))(((object,looping over)))What if we want to find all the events for which we have stored a coefficient? The properties don't form a predictable series, like they would in an array, so we cannot use a normal `for` loop. JavaScript provides a loop construct specifically for going over the properties of an object. It looks a little like a normal `for` loop but distinguishes itself by the use of the word `in`.

```
for (var event in map)
  console.log("The correlation for '" + event +
              "' is " + map[event]);
// → The correlation for 'pizza' is 0.069
// → The correlation for 'touched tree' is -0.081
```

[[analysis]] == The final analysis ==

(((journal)))(((weresquirrel example)))(((gatherCorrelations function)))To find all the types of events that are present in the data set, we simply process each entry in turn and then loop over the events in that entry. We keep an object `phis` that has correlation coefficients for all the event types we have seen so far. Whenever we run across a type that isn't in the `phis` object yet, we compute its correlation and add it to the object.

// test: clip // include_code strip_log

```
function gatherCorrelations(journal) {
  var phis = {};
  for (var entry = 0; entry < journal.length; entry++) {
    var events = journal[entry].events;
    for (var i = 0; i < events.length; i++) {
      var event = events[i];
      if (!(event in phis))
        phis[event] = phi(tableFor(event, journal));
    }
  }
  return phis;
}

var correlations = gatherCorrelations(JOURNAL);
console.log(correlations.pizza);
// → 0.068599434
```

(((correlation)))Let's see what came out.

// test: no

```
for (var event in correlations)
  console.log(event + ": " + correlations[event]);
// → carrot:   0.0140970969
// → exercise: 0.0685994341
// → weekend:  0.1371988681
// → bread:   -0.0757554019
// → pudding: -0.0648203724
// and so on...
```

(((for/in loop)))Most correlations seem to lie close to zero. Eating carrots, bread, or pudding apparently does not trigger squirrel-lycanthropy. It *does* seem to occur somewhat more often on weekends, however. Let's filter the results to show only correlations greater than 0.1 or less than -0.1.

// start_code // test: no

```
for (var event in correlations) {
  var correlation = correlations[event];
  if (correlation > 0.1 || correlation < -0.1)
    console.log(event + ": " + correlation);
}
// → weekend:        0.1371988681
// → brushed teeth: -0.3805211953
// → candy:          0.1296407447
// → work:          -0.1371988681
// → spaghetti:      0.2425356250
// → reading:        0.1106828054
// → peanuts:        0.5902679812
```

A-ha! There are two factors whose ((correlation)) is clearly stronger than the others. Eating ((peanuts)) has a strong positive effect on the chance of turning into a squirrel, whereas brushing his teeth has a significant negative effect.

Interesting. Let's try something.

// include_code strip_log

```
for (var i = 0; i < JOURNAL.length; i++) {
  var entry = JOURNAL[i];
  if (hasEvent("peanuts", entry) &&
     !hasEvent("brushed teeth", entry))
    entry.events.push("peanut teeth");
}
console.log(phi(tableFor("peanut teeth", JOURNAL)));
// → 1
```

Well, that's unmistakable! The phenomenon occurs precisely when Jacques eats ((peanuts)) and fails to brush his teeth. If only he weren't such a slob about dental hygiene, he'd have never even noticed his affliction.

Knowing this, Jacques simply stops eating peanuts altogether and finds that this completely puts an end to his transformations.

(((weresquirrel example)))All is well with Jacques for a while. But a few years later, he loses his ((job)) and is eventually forced to take employment with a ((circus)), where he performs as *The Incredible Squirrelman* by stuffing his mouth with peanut butter before every show.

One day, fed up with this pitiful existence, Jacques fails to change back into his human form, hops through a crack in the circus tent, and vanishes into the forest. He is never seen again.

== Further arrayology ==

(((array,methods)))(((method)))Before finishing up this chapter, I want to introduce you to a few more object-related concepts. We'll start by introducing some generally useful array methods.

(((push method)))(((pop method)))(((shift method)))(((unshift method)))We saw `push` and `pop`, which add and remove elements at the end of an array, link:04_data.html#array_methods[earlier] in this chapter. The corresponding methods for adding and removing things at the start of an array are called `unshift` and `shift`.

```
var todoList = [];
function rememberTo(task) {
  todoList.push(task);
}
function whatIsNext() {
  return todoList.shift();
}
function urgentlyRememberTo(task) {
  todoList.unshift(task);
}
```

(((task management example)))The previous program manages lists of tasks. You add tasks to the end of the list by calling `rememberTo("eat")`, and when you're ready to do something, you call `whatIsNext()` to get (and remove) the front item from the list. The `urgentlyRememberTo` function also adds a task but adds it to the front instead of the back of the list.

(((array,searching)))(((indexOf method)))(((lastIndexOf method)))The `indexOf` method has a sibling called `lastIndexOf`, which starts searching for the given element at the end of the array instead of the front.

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

Both `indexOf` and `lastIndexOf` take an optional second argument that indicates where to start searching from.

(((slice method)))(((array,indexing)))Another fundamental method is `slice`, which takes a start index and an end index and returns an array that has only the elements between those indices. The start index is inclusive, the end index exclusive.

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

(((string,indexing)))When the end index is not given, `slice` will take all of the elements after the start index. Strings also have a `slice` method, which has a similar effect.

(((concatenation)))(((concat method)))The `concat` method can be used to glue arrays together, similar to what the `+` operator does for strings. The following example shows both `concat` and `slice` in action. It takes an array and an index, and it returns a new array that is a copy of the original array with the element at the given index removed.

```
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"]
```

== Strings and their properties ==

(((string,properties)))We can read properties like `length` and `toUpperCase` from string values. But if you try to add a new property, it doesn't stick.

```
var myString = "Fido";
myString.myProperty = "value";
console.log(myString.myProperty);
// → undefined
```

Values of type string, number, and Boolean are not objects, and though the language doesn't complain if you try to set new properties on them, it doesn't actually store those properties. The values are immutable and cannot be changed.

(((string,methods)))(((slice method)))(((indexOf method)))(((string,searching)))But these types do have some built-in properties. Every string value has a number of methods. The most useful ones are probably `slice` and `indexOf`, which resemble the array methods of the same name.

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

One difference is that a string's `indexOf` can take a string containing more than one character, whereas the corresponding array method looks only for a single element.

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

(((whitespace)))(((trim method)))The `trim` method removes whitespace (spaces, newlines, tabs, and similar characters) from the start and end of a string.

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

(((length property,for string)))(((charAt method)))(((string,indexing)))We have already seen the string type's `length` property. Accessing the individual characters in a string can be done with the `charAt` method but also by simply reading numeric properties, like you'd do for an array.

```
var string = "abc";
console.log(string.length);
// → 3
console.log(string.charAt(0));
// → a
console.log(string[1]);
// → b
```

[[arguments_object]] == The arguments object ==

(((arguments object)))(((length property)))(((parameter)))(((optional argument)))(((array-like object)))Whenever a function is called, a special variable named `arguments` is added to the environment in which the function body runs. This variable refers to an object that holds all of the arguments passed to the function. Remember that in JavaScript you are allowed to pass more (or fewer) arguments to a function than the number of parameters the function itself declares.

```
function noArguments() {}
noArguments(1, 2, 3); // This is okay
function threeArguments(a, b, c) {}
threeArguments(); // And so is this
```

(((length property)))The `arguments` object has a `length` property that tells us the number of arguments that were really passed to the function. It also has a property for each argument, named 0, 1, 2, and so on.

indexsee:[pseudo array,array-like object] (((array,methods)))If that sounds a lot like an array to you, you're right, it *is* a lot like an array. But this object, unfortunately, does not have any array methods (like `slice` or `indexOf`), so it is a little harder to use than a real array.

```
function argumentCounter() {
  console.log("You gave me", arguments.length, "arguments.");
}
argumentCounter("Straw man", "Tautology", "Ad hominem");
// → You gave me 3 arguments.
```

(((journal)))(((console.log)))(((variadic function)))Some functions can take any number of arguments, like `console.log`. These typically loop over the values in their `arguments` object. They can be used to create very pleasant interfaces. For example, remember how we created the entries to Jacques' journal.

```
addEntry(["work", "touched tree", "pizza", "running",
          "television"], false);
```

Since he is going to be calling this function a lot, we could create an alternative that is easier to call.

```
function addEntry(squirrel) {
  var entry = {events: [], squirrel: squirrel};
  for (var i = 1; i < arguments.length; i++)
    entry.events.push(arguments[i]);
  journal.push(entry);
}
addEntry(true, "work", "touched tree", "pizza",
         "running", "television");
```

(((arguments object,indexing)))This version reads its first argument (`squirrel`) in the normal way and then goes over the rest of the arguments (the loop starts at index 1, skipping the first) to gather them into an array.

== The Math object ==

(((Math object)))(((Math.min function)))(((Math.max function)))(((Math.sqrt function)))(((minimum)))(((maximum)))(((square root)))As we've seen, `Math` is a grab-bag of number-related utility functions, such as `Math.max` (maximum), `Math.min` (minimum), and `Math.sqrt` (square root).

[[namespace*pollution]] (((namespace)))(((namespace pollution)))(((object)))The* `Math` *object is used simply as a container to group a bunch of related functionality. There is only one* `Math` *object, and it is almost never useful as a value. Rather, it provides a _namespace* so that all these functions and values do not have to be global variables.

(((variable,naming)))Having too many global variables “pollutes” the namespace. The more names that have been taken, the more likely you are to accidentally overwrite the value of some variable. For example, it's not unlikely that you'll want to name something `max` in one of your programs. Since JavaScript's built-in `max` function is tucked safely inside the `Math` object, we don't have to worry about overwriting it.

Many languages will stop you, or at least warn you, when you are defining a variable with a name that is already taken. JavaScript does neither, so be careful.

(((Math.cos function)))(((Math.sin function)))(((Math.tan function)))(((Math.acos function)))(((Math.asin function)))(((Math.atan function)))(((Math.PI constant)))(((cosine)))(((sine)))(((tangent)))(((PI constant)))(((pi)))Back to the `Math` object. If you need to do ((trigonometry)), `Math` can help. It contains `cos` (cosine), `sin` (sine), and `tan` (tangent), as well as their inverse functions, `acos`, `asin`, and `atan`, respectively. The number π (pi)—or at least the closest approximation that fits in a JavaScript number—is available as `Math.PI`. (There is an old programming tradition of writing the names of ((constant)) values in all caps.)

// test: no

```
function randomPointOnCircle(radius) {
  var angle = Math.random() * 2 * Math.PI;
  return {x: radius * Math.cos(angle),
          y: radius * Math.sin(angle)};
}
console.log(randomPointOnCircle(2));
// → {x: 0.3667, y: 1.966}
```

If sines and cosines are not something you are very familiar with, don't worry. When they are used in this book, in link:13_dom.html#sin_cos[Chapter 13], I'll explain them.

(((Math.random function)))(((random number)))The previous example uses `Math.random`. This is a function that returns a new pseudorandom number between zero (inclusive) and one (exclusive) every time you call it.

// test: no

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

(((pseudorandom number)))(((random number)))Though computers are deterministic machines—they always react the same way if given the same input—it is possible to have them produce numbers that appear random. To do this, the machine keeps a number (or a bunch of numbers) in its internal state. Then, every time a random number is requested, it performs some complicated deterministic computations on this internal state and returns part of the result of those computations. The machine also uses the outcome to change its own internal state so that the next “random” number produced will be different.

(((rounding)))(((Math.floor function)))If we want a whole random number instead of a fractional one, we can use `Math.floor` (which rounds down to the nearest whole number) on the result of `Math.random`.

// test: no

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

Multiplying the random number by 10 gives us a number greater than or equal to zero, and below 10. Since `Math.floor` rounds down, this expression will produce, with equal chance, any number from 0 through 9.

(((Math.ceil function)))(((Math.round function)))There are also the functions `Math.ceil` (for “ceiling”, which rounds up to a whole number) and `Math.round` (to the nearest whole number).

== The global object ==

(((global object)))(((window variable)))(((global scope)))(((scope)))(((object)))The global scope, the space in which global variables live, can also be approached as an object in JavaScript. Each global variable is present as a ((property)) of this object. In ((browser))s, the global scope object is stored in the `window` variable.

// test: no

```
var myVar = 10;
console.log("myVar" in window);
// → true
console.log(window.myVar);
// → 10
```

== Summary ==

Objects and arrays (which are a specific kind of object) provide ways to group several values into a single value. Conceptually, this allows us to put a bunch of related things in a bag and run around with the bag, instead of trying to wrap our arms around all of the individual things and trying to hold on to them separately.

Most values in JavaScript have properties, the exceptions being `null` and `undefined`. Properties are accessed using `value.propName` or `value["propName"]`. Objects tend to use names for their properties and store more or less a fixed set of them. Arrays, on the other hand, usually contain varying numbers of conceptually identical values and use numbers (starting from 0) as the names of their properties.

There *are* some named properties in arrays, such as `length` and a number of methods. Methods are functions that live in properties and (usually) act on the value they are a property of.

Objects can also serve as maps, associating values with names. The `in` operator can be used to find out whether an object contains a property with a given name. The same keyword can also be used in a `for` loop (`for (var name in object)`) to loop over an object's properties.

== Exercises ==

=== The sum of a range ===

(((summing (exercise))))The link:00_intro.html#intro[introduction] of this book alluded to the following as a nice way to compute the sum of a range of numbers:

// test: no

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

(((range function)))(((sum function)))Write a `range` function that takes two arguments, `start` and `end`, and returns an array containing all the numbers from `start` up to (and including) `end`.

Next, write a `sum` function that takes an array of numbers and returns the sum of these numbers. Run the previous program and see whether it does indeed return 55.

(((optional argument)))As a bonus assignment, modify your `range` function to take an optional third argument that indicates the “step” value used to build up the array. If no step is given, the array elements go up by increments of one, corresponding to the old behavior. The function call `range(1, 10, 2)` should return `[1, 3, 5, 7, 9]`. Make sure it also works with negative step values so that `range(5, 2, -1)` produces `[5, 4, 3, 2]`.

ifdef::interactive_target[]

// test: no

```
// Your code here.

console.log(range(1, 10));
// → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
console.log(range(5, 2, -1));
// → [5, 4, 3, 2]
console.log(sum(range(1, 10)));
// → 55
```

endif::interactive_target[]

!!hint!!

(((summing (exercise))))(((array,creation)))(((square brackets)))Building up an array is most easily done by first initializing a variable to `[]` (a fresh, empty array) and repeatedly calling its `push` method to add a value. Don't forget to return the array at the end of the function.

(((array,indexing)))(((comparison)))Since the end boundary is inclusive, you'll need to use the `<=` operator rather than simply `<` to check for the end of your loop.

(((arguments object)))To check whether the optional step argument was given, either check `arguments.length` or compare the value of the argument to `undefined`. If it wasn't given, simply set it to its ((default value)) (1) at the top of the function.

(((range function)))(((for loop)))Having `range` understand negative step values is probably best done by writing two separate loops—one for counting up and one for counting down—because the comparison that checks whether the loop is finished needs to be `>=` rather than `<=` when counting downward.

It might also be worthwhile to use a different default step, namely, -1, when the end of the range is smaller than the start. That way, `range(5, 2)` returns something meaningful, rather than getting stuck in an ((infinite loop)).

!!hint!!

=== Reversing an array ===

(((reversing (exercise))))(((reverse method)))(((array,methods)))Arrays have a method `reverse`, which changes the array by inverting the order in which its elements appear. For this exercise, write two functions, `reverseArray` and `reverseArrayInPlace`. The first, `reverseArray`, takes an array as argument and produces a *new* array that has the same elements in the inverse order. The second, `reverseArrayInPlace`, does what the `reverse` method does: it modifies the array given as argument in order to reverse its elements. Neither may use the standard `reverse` method.

(((efficiency)))(((pure function)))(((side effect)))Thinking back to the notes about side effects and pure functions in the link:03_functions.html#pure[previous chapter], which variant do you expect to be useful in more situations? Which one is more efficient?

ifdef::interactive_target[]

// test: no

```
// Your code here.

console.log(reverseArray(["A", "B", "C"]));
// → ["C", "B", "A"];
var arrayValue = [1, 2, 3, 4, 5];
reverseArrayInPlace(arrayValue);
console.log(arrayValue);
// → [5, 4, 3, 2, 1]
```

endif::interactive_target[]

!!hint!!

(((reversing (exercise))))There are two obvious ways to implement `reverseArray`. The first is to simply go over the input array from front to back and use the `unshift` method on the new array to insert each element at its start. The second is to loop over the input array backward and use the `push` method. Iterating over an array backward requires a (somewhat awkward) `for` specification like `(var i = array.length - 1; i >= 0; i--)`.

Reversing the array in place is harder. You have to be careful not to overwrite elements that you will later need. Using `reverseArray` or otherwise copying the whole array (`array.slice(0)` is a good way to copy an array) works but is cheating.

The trick is to *swap* the first and last elements, then the second and second-to-last, and so on. You can do this by looping over half the length of the array (use `Math.floor` to round down—you don't need to touch the middle element in an array with an odd length) and swapping the element at position `i` with the one at position `array.length - 1 - i`. You

can use a local variable to briefly hold on to one of the elements, overwrite that one with its mirror image, and then put the value from the local variable in the place where the mirror image used to be.

!!hint!!

[[list]] === A list ===

(((data structure)))(((list (exercise))))(((linked list)))(((object)))(((array)))(((collection)))Objects, as generic blobs of values, can be used to build all sorts of data structures. A common data structure is the *list* (not to be confused with the array). A list is a nested set of objects, with the first object holding a reference to the second, the second to the third, and so on.

// include_code

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

The resulting objects form a chain, like this:

image::img/linked-list.svg[alt="A linked list",width="6cm"]

(((structure sharing)))(((memory)))A nice thing about lists is that they can share parts of their structure. For example, if I create two new values `{value: 0, rest: list}` and `{value: -1, rest: list}` (with `list` referring to the variable defined earlier), they are both independent lists, but they share the structure that makes up their last three elements. In addition, the original list is also still a valid three-element list.

Write a function `arrayToList` that builds up a data structure like the previous one when given `[1, 2, 3]` as argument, and write a `listToArray` function that produces an array from a list. Also write the helper functions `prepend`, which takes an element and a list and creates a new list that adds the element to the front of the input list, and `nth`, which takes a list and a number and returns the element at the given position in the list, or `undefined` when there is no such element.

(((recursion)))If you haven't already, also write a recursive version of `nth`.

ifdef::interactive_target[]

// test: no

```
// Your code here.

console.log(arrayToList([10, 20]));
// → {value: 10, rest: {value: 20, rest: null}}
console.log(listToArray(arrayToList([10, 20, 30])));
// → [10, 20, 30]
console.log(prepend(10, prepend(20, null)));
// → {value: 10, rest: {value: 20, rest: null}}
console.log(nth(arrayToList([10, 20, 30]), 1));
// → 20
```

endif::interactive_target[]

!!hint!!

(((list (exercise))))(((linked list)))Building up a list is best done back to front. So `arrayToList` could iterate over the array backward (see previous exercise) and, for each element, add an object to the list. You can use a local variable to hold the part of the list that was built so far and use a pattern like `list = {value: X, rest: list}` to add an element.

(((for loop)))To run over a list (in `listToArray` and `nth`), a `for` loop specification like this can be used:

```
for (var node = list; node; node = node.rest) {}
```

Can you see how that works? Every iteration of the loop, `node` points to the current sublist, and the body can read its `value` property to get the current element. At the end of an iteration, `node` moves to the next sublist. When that is null, we have reached the end of the list and the loop is finished.

(((recursion)))The recursive version of `nth` will, similarly, look at an ever smaller part of the "tail" of the list and at the same time count down the index until it reaches zero, at which point it can return the `value` property of the node it is looking at. To get the zeroeth element of a list, you simply take the `value` property of its head node. To get element *N* + 1, you take the **N**th element of the list that's in this list's `rest` property.

!!hint!!

[[exercise_deep_compare]] === Deep comparison ===

(((deep comparison (exercise))))(((comparison)))(((deep comparison)))(((== operator)))The `==` operator compares objects by identity. But sometimes, you would prefer to compare the values of their actual properties.

Write a function, `deepEqual`, that takes two values and returns true only if they are the same value or are objects with the same properties whose values are also equal when compared with a recursive call to `deepEqual`.

(((null)))(((=== operator)))(((typeof operator)))To find out whether to compare two things by identity (use the `===` operator for that) or by looking at their properties, you can use the `typeof` operator. If it produces `"object"` for both values, you should do a deep comparison. But you have to take one silly exception into account: by a historical accident, `typeof null` also produces `"object"`.

ifdef::interactive_target[]

// test: no

```
// Your code here.

var obj = {here: {is: "an"}, object: 2};
console.log(deepEqual(obj, obj));
// → true
console.log(deepEqual(obj, {here: 1, object: 2}));
// → false
console.log(deepEqual(obj, {here: {is: "an"}, object: 2}));
// → true
```

endif::interactive_target[]

!!hint!!

(((deep comparison (exercise)))(((typeof operator)))(((object)))(((=== operator)))Your test for whether you are dealing with a real object will look something like `typeof x == "object" && x != null`. Be careful to compare properties only when *both* arguments are objects. In all other cases you can just immediately return the result of applying `===`.

(((for/in loop)))(((in operator)))Use a `for` / `in` loop to go over the properties. You need to test whether both objects have the same set of property names and whether those properties have identical values. The first test can be done by counting the properties in both objects and returning false if the numbers of properties are different. If they're the same, then go over the properties of one object, and for each of them, verify that the other object also has the property. The values of the properties are compared by a recursive call to `deepEqual`.

(((return value)))Returning the correct value from the function is best done by immediately returning false when a mismatch is noticed and returning true at the end of the function.

!!hint!!

Funciones de Order Superior

> Tzu-li y Tzu-ssu estaban presumiendo acerca del tamaño de sus úlitmos programas. 'Doscientas mil líneas,' dijo Tzu-li, '¡sin contar los comentarios!'. Tzu-ssu respondió, 'Pssh, el mío tiene ya casi *un millión* de líneas.' El Maestro Yuan-Ma dijo, 'Mi mejor programa tiene quinientas líneas'. Oyendo esto, Tzu-li y Tzu-ssu fueron iluminados.
> Master Yuan-Ma, The Book of Programming
>
> Hay dos formas de construir un diseño de software: Una forma es hacerlo tan simple que obviamente no tenga deficiencias, y la otra forma es hacerlo tan complicado que no haya obvias deficiecias. C.A.R. Hoare, 1980 ACM Turing Award Lecture

Un programa grande es costoso y no sólo por el tiempo que toma construirlo. El tamaño casi siempre involucra *complejidad*, y la complejidad confunde a los programadores. Los programadores confundidos, a su vez, tienden a introducir errores (*bugs*) en los programas. Un programa grande, además, da un amplio espacio para que estos bugs se oculten, haciéndolos difíciles de encontrar.

Regresemos brevemente a los dos programas finales en la introducción. El primero es auto-contenido y tiene seis líneas de longitud.

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

El segundo depende de dos funciones externas y es una sola línea

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

¿Cuál de los dos es más probable que tenga un bug?

Si contamos el tamaño de definición de `suma` y `rango`, el segungo programa también es grande–incluso más grande que el primero. Pero aún así, yo diría que es más probable que sea correcto.

Es más probable que sea correcto porque la solución es expresada en un *vocabulario* que corresponde al problema que está siendo resuelto. Sumar un rango de enteros no tiene que ver con bucles y contadores. Es acerca de rangos y sumas.

Las definiciones de este vocabulario (las funciones `suma` y `rango`) todavía incluirán bucles, contadores y otros detalles incidentales. Pero debido a que están expresando conceptos más simples que el programa como un todo, es más fácil acertar.

Abstracción

En el contexto de la programación, vocabularios de este estilo son usualmente llamados *abstracción*. Las abstracciones econden detalles y nos dan la habilidad de hablar de los problemas en un nivel más alto (o más abstracto).

Como una analogía, compara estas dos recetas para la sopa de guisantes:

`Pon una 1 taza de guisantes secos por persona en un contenedor. Agrega agua hasta que los guisantes estén cubiertos. Deja los guisantes en el agua por lo menos 12 horas. Saca los guisantes del agua y ponlos en en un sartén para cocer. Agrega 4 copas de agua por persona. Cubre el sartén y mantén los hirviéndose a fuego lento por dos horas. Agrega la mitad de una cebolla por persona. Córtala en piezas con un cuchillo. Agrégalas a los guisantes. Toma un diente de ajo por persona. Córtalos en piezas con un cuchillo. Agrégalos a los guisantes. Toma una zanahoria por persona. Córtalas en piezas. ¡Con un cuchillo! Agrégalas a los guisantes. Cocina por 10

minutos más.`

Y la segunda receta:

`Por persona: 1 taza de guisantes partidos secos, media cebolla picada, un diente de ajo y una zanohoria.

Remoja los guisantes por 12 horas Soak peas for 12 hours. Hierve a fuego lento por 2 horas en

4 tazas(por persona). Pica y agrega los vegetales. Cocina por 10 minutos más.`

La segunda es más corta y más fácil de interpretar. Pero necesitas entender unos cuántas palabras relacionadas con la cocina-**remojar**, **picar** y, imagino, **vegetal**.

Cuando programamos, no podemos confiar en que todas las palabras que necesitamos estén esperándonos en el diccionario. Así que podrías caer en el patrón de la primera receta–trabajar en los pasos precisos que la computadora debe realizar, uno por uno, sin ver los conceptos de alto nivel que estos expresan.

Se tiene que convertir en una segunda naturaleza, para un programador, notar cuando un concepto está rogando ser abstraído en una nueva palabra.

Abstrayendo transversal de array

Las fuunciones planas, como hemos visto hasta ahora, son una buena forma de construir abstracciones. Pero algunas veces se quedan cortas.

En el capítulo anterior], este tipo de *bucle* `for` apareció varias veces:

```
var array = [1, 2, 3];
for (var i = 0; i < array.length; i++) {
  var actual = array[i];
  console.log(actual);
}
```

Está tratando de decir, "Cada elemento, escríbelo en la consola". Pero usa una forma rebuscada que implica una variable `i`, una revisión del tamaño del array y una declaración de variable extra para guardar el elemento actual. A parte de causar un poco de dolor de ojos, esto nos da mucho espacio para errores potenciales. Podríamos accidentalmente reusar la variable `i`, escribir mal `length` como `lenght`, confundir las variables `i` y `actual` y así por el estilo. Así que tratemos de abstraer esto en una función. ¿Puedes pensar en alguna forma?

Bueno, es fácil escribir una función que vaya a través de un array y llamar `console.log` en cada elemento.

```
function logEach(array) {
  for (var i = 0; i < array.length; i++)
    console.log(array[i]);
}
```

Pero, ¿qué pasa si queremos hacer otra cosa que loggear los elementos? Debido a que "hacer algo" puede ser representado como una función y las funciones son sólo valores, podemos pasar nuestra acción como un valor función.

```
function forEach(array, accion) {
  for (var i = 0; i < array.length; i++)
    accion(array[i]);
}

forEach(["Wampeter", "Foma", "Granfalloon"], console.log);
// → Wampeter
// → Foma
// → Granfalloon
```

En algunos navegadores llamar a `console.log` de esta manera no funcionara. Puedes usar `alert` en lugar de `console.log` si este ejemplo falla.

A menudo, no le pasas una función predefinida a `forEach` sino que creas una función en el acto.

```
var numeros = [1, 2, 3, 4, 5], suma = 0;
forEach(numeros, function(numero) {
  suma += numero;
});
console.log(suma);
// → 15
```

Esto luce muy parecido al clásico bucle `for` , con su cuerpo escrito debajo de él. Sin embargo, ahora el cuerpo está dentro del valor función, así como dentro de los *paréntesis* de la llamada a `forEach` . Esta es la razón de que tenga que ser terminado con una llave *y* un paréntesis de cierre.

Usando este patrón, podemos especificar un nombre de variable para el elemento actual(`numero`), en vez de tener que tomarlo del array manuelmente.

De hecho, no necesitamos escribir `forEach` nosotros mismos. Está disponible como un método estándar en los arrays. Debido a que el array le es pasado como el elemento sobre el que actúa el método, `forEach` sólo recibe un argumento requerido: la función que será ejecutada para cada elemento.

Para ilustrar lo útil que esto es, miremos otra vez la función del link:04_data.html#analysis[capítulo anterior]. Contiene dos bucles que recorren un arreglo.

```
function reuneCorrelaciones(diario) {
  var phis = {};
  for (var entrada = 0; entrada < diario.length; entrada++) {
    var eventos = diario[entrada].events;
    for (var i = 0; i < events.length; i++) {
      var evento = eventos[i];
      if (!(evento in phis))
        phis[evento] = phi(tableFor(evento, diario));
    }
  }
  return phis;
}
```

(((método forEach))) `forEach` la hace un poco más corta y limpia.

```
function reuneCorrelaciones(diario) {
  var phis = {};
  diario.forEach(function(entrada) {
    entrada.eventos.forEach(function(evento) {
      if (!(evento in phis))
        phis[evento] = phi(tableFor(evento, diario));
    });
  });
  return phis;
}
```

== Funciones de Orden Superior ==

(((función,de orden superior)))(((función,como valor)))Las funciones que operan en otras funceiones, ya sea tomándolas como argumentos o regresándolas, son llamadas *funciones de orden superior*. Si ya has aceptado el hecho de que las funciones son valores reulares, no hay nada de especial en el hecho de que estas funciones existan. El términos viene de las ((matemáticas)), en dónde la distinción entre las funciones y otros valores es tomado más seriamente.

(((abstracción)))Las funciones de orden superior nos permiten abstraer *acciones*, no sólo valores. Pueden venir en diferentes formas. Por ejemplo puedes tener funciones que creen nuevas funciones.

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

Y puedes tener funciones que cambien otras funciones.

```
function ruidosa(f) {
  return function(arg) {
    console.log("llamando con", arg);
    var val = f(arg);
    console.log("llamada con", arg, "- got", val);
    return val;
  };
}
ruidosa(Boolean)(0);
// → llamando con 0
// → llamada con 0 - obtuve false
```

Incluso puedes escribir funciones que creen nuevos tipos ((control de flujo)).

```
function a_menos_que(condicion, entonces) {
  if (!condicion) entonces();
}
function repetir(veces, cuerpo) {
  for (var i = 0; i < veces; i++) cuerpo(i);
}

repetir(3, function(n) {
  a_menos_que(n % 2, function() {
    console.log(n, "es par");
  });
});
// → 0 es par
// → 2 es par
```

(((inner function)))(((nesting,of functions)))((({} (block))))(((local variable)))(((closure)))The ((lexical scoping)) rules that we discussed in link:03_functions.html#scoping[Chapter 3] work to our advantage when using functions in this way. In the previous example, the `n` variable is a ((parameter)) to the outer function. Because the inner function lives inside the environment of the outer one, it can use `n`. The bodies of such inner functions can access the variables around them. They can play a role similar to the `{}` blocks used in regular loops and conditional statements. An important difference is that variables declared inside inner functions do not end up in the environment of the outer function. And that is usually a good thing.

== Passing along arguments ==

(((function,wrapping)))(((arguments object)))The `noisy` function defined earlier, which wraps its argument in another function, has a rather serious deficit.

```
function noisy(f) {
  return function(arg) {
    console.log("calling with", arg);
    var val = f(arg);
    console.log("called with", arg, "- got", val);
    return val;
  };
}
```

If `f` takes more than one ((parameter)), it gets only the first one. We could add a bunch of arguments to the inner function (`arg1` , `arg2` , and so on) and pass them all to `f` , but it is not clear how many would be enough. This solution would also deprive `f` of the information in `arguments.length` . Since we'd always pass the same number of arguments, it wouldn't know how many arguments were originally given.

(((apply method)))(((array-like object)))(((function,application)))For these kinds of situations, JavaScript functions have an `apply` method. You pass it an array (or array-like object) of arguments, and it will call the function with those arguments.

```
function transparentWrapping(f) {
  return function() {
    return f.apply(null, arguments);
  };
}
```

(((null)))That's a useless function, but it shows the pattern we are interested in—the function it returns passes all of the given arguments, and only those arguments, to `f` . It does this by passing its own `arguments` object to `apply` . The first argument to `apply` , for which we are passing `null` here, can be used to simulate a ((method)) call. We will come back to that in the link:06_object.html#call_method[next chapter].

== JSON ==

(((array)))(((function,higher-order)))(((forEach method)))(((data set)))Higher-order functions that somehow apply a function to the elements of an array are widely used in JavaScript. The `forEach` method is the most primitive such function. There are a number of other variants available as methods on arrays. To familiarize ourselves with them, let's play around with another data set.

(((ancestry example)))A few years ago, someone crawled through a lot of archives and put together a book on the history of my family name (Haverbeke—meaning Oatbrook). I opened it hoping to find knights, pirates, and alchemists ... but the book turns out to be mostly full of Flemish ((farmer))s. For my amusement, I extracted the information on my direct ancestors and put it into a computer-readable format.

(((data format)))(((JSON)))The file I created looks something like this:

[source,application/json]

```
[
  {"name": "Emma de Milliano", "sex": "f",
   "born": 1876, "died": 1956,
   "father": "Petrus de Milliano",
   "mother": "Sophia van Damme"},
  {"name": "Carolus Haverbeke", "sex": "m",
   "born": 1832, "died": 1905,
   "father": "Carel Haverbeke",
   "mother": "Maria van Brussel"},
  … and so on
]
```

indexsee:JavaScript Object Notation,JSON))This format is called JSON (pronounced “Jason”), which stands for JavaScript Object Notation. It is widely used as a data storage and communication format on the Web.

(((array)))(((object)))(((quoting,in JSON)))(((comment)))JSON is similar to JavaScript's way of writing arrays and objects, with a few restrictions. All property names have to be surrounded by double quotes, and only simple data expressions are allowed—no function calls, variables, or anything that involves actual computation. Comments are not allowed in JSON.

(((JSON.stringify function)))(((JSON.parse function)))(((serialization)))(((deserialization)))(((parsing)))JavaScript gives us functions, `JSON.stringify` and `JSON.parse`, that convert data from and to this format. The first takes a JavaScript value and returns a JSON-encoded string. The second takes such a string and converts it to the value it encodes.

```
var string = JSON.stringify({name: "X", born: 1980});
console.log(string);
// → {"name":"X","born":1980}
console.log(JSON.parse(string).born);
// → 1980
```

(((ANCESTRY*FILE data set)))*The variable `ANCESTRY_FILE`, *available in the ((sandbox)) for this chapter and in http://eloquentjavascript.net/code/ancestry.js[a downloadable file] on the website(!book (http://eloquentjavascript.net/code#5[_eloquentjavascript.net/code#5]*)!), contains the content of my ((JSON)) file as a string. Let's decode it and see how many people it contains.

// include_code strip_log

```
var ancestry = JSON.parse(ANCESTRY_FILE);
console.log(ancestry.length);
// → 39
```

== Filtering an array ==

(((array,methods)))(((array,filtering)))(((filter method)))(((function,higher-order)))(((predicate function)))To find the people in the ancestry data set who were young in 1924, the following function might be helpful. It filters out the elements in an array that don't pass a test.

```
function filter(array, test) {
  var passed = [];
  for (var i = 0; i < array.length; i++) {
    if (test(array[i]))
      passed.push(array[i]);
  }
  return passed;
}

console.log(filter(ancestry, function(person) {
  return person.born > 1900 && person.born < 1925;
}));
// → [{name: "Philibert Haverbeke", …}, …]
```

(((function,as value)))(((function,application)))This uses the argument named `test`, a function value, to fill in a “gap” in the computation. The `test` function is called for each element, and its return value determines whether an element is included in the returned array.

(((ancestry example)))Three people in the file were alive and young in 1924: my grandfather, grandmother, and great-aunt.

(((filter method)))(((pure function)))(((side effect)))Note how the `filter` function, rather than deleting elements from the existing array, builds up a new array with only the elements that pass the test. This function is *pur*e. It does not modify the array it is given.

Like `forEach`, `filter` is also a ((standard)) method on arrays. The example defined the function only in order to show what it does internally. From now on, we'll use it like this instead:

```
console.log(ancestry.filter(function(person) {
  return person.father == "Carel Haverbeke";
}));
// → [{name: "Carolus Haverbeke", …}]
```

== Transforming with map ==

(((array,methods)))(((map method)))(((ancestry example)))Say we have an array of objects representing people, produced by filtering the `ancestry` array somehow. But we want an array of names, which is easier to read.

(((function,higher-order)))The `map` method transforms an array by applying a function to all of its elements and building a new array from the returned values. The new array will have the same length as the input array, but its content will have been “mapped” to a new form by the function.

// test: join

```
function map(array, transform) {
  var mapped = [];
  for (var i = 0; i < array.length; i++)
    mapped.push(transform(array[i]));
  return mapped;
}

var overNinety = ancestry.filter(function(person) {
  return person.died - person.born > 90;
});
console.log(map(overNinety, function(person) {
  return person.name;
}));
// → ["Clara Aernoudts", "Emile Haverbeke",
//    "Maria Haverbeke"]
```

Interestingly, the people who lived to at least 90 years of age are the same three people who we saw before—the people who were young in the 1920s, which happens to be the most recent generation in my data set. I guess ((medicine)) has come a long way.

Like `forEach` and `filter`, `map` is also a standard method on arrays.

== Summarizing with reduce ==

(((array,methods)))(((summing example)))(((reduce method)))(((ancestry example)))Another common pattern of computation on arrays is computing a single value from them. Our recurring example, summing a collection of numbers, is an instance of this. Another example would be finding the person with the earliest year of birth in the data set.

(((function,higher-order)))(((fold function)))The higher-order operation that represents this pattern is called *reduce* (or sometimes *fold*). You can think of it as folding up the array, one element at a time. When summing numbers, you'd start with the number zero and, for each element, combine it with the current sum by adding the two.

The parameters to the `reduce` function are, apart from the array, a combining function and a start value. This function is a little less straightforward than `filter` and `map`, so pay careful attention.

```
function reduce(array, combine, start) {
  var current = start;
  for (var i = 0; i < array.length; i++)
    current = combine(current, array[i]);
  return current;
}

console.log(reduce([1, 2, 3, 4], function(a, b) {
  return a + b;
}, 0));
// → 10
```

(((reduce method)))The standard array method `reduce`, which of course corresponds to this function, has an added convenience. If your array contains at least one element, you are allowed to leave off the `start` argument. The method will take the first element of the array as its start value and start reducing at the second element.

(((ancestry example)))(((minimum)))To use `reduce` to find my most ancient known ancestor, we can write something like this:

// test: no

```
console.log(ancestry.reduce(function(min, cur) {
  if (cur.born < min.born) return cur;
  else return min;
}));
// → {name: "Pauwels van Haverbeke", born: 1535, …}
```

== Composability ==

(((loop)))(((minimum)))(((ancestry example)))Consider how we would have written the previous example (finding the person with the earliest year of birth) without higher-order functions. The code is not that much worse.

// test: no

```
var min = ancestry[0];
for (var i = 1; i < ancestry.length; i++) {
  var cur = ancestry[i];
  if (cur.born < min.born)
    min = cur;
}
console.log(min);
// → {name: "Pauwels van Haverbeke", born: 1535, …}
```

There are a few more ((variable))s, and the program is two lines longer but still quite easy to understand.

[[average*function]] (((average function)))(((composability)))(((function,higher-order)))Higher-order functions start to shine when you need to _compose* functions. As an example, let's write code that finds the average age for men and for women in the data set.

// test: clip

```
function average(array) {
  function plus(a, b) { return a + b; }
  return array.reduce(plus) / array.length;
}
function age(p) { return p.died - p.born; }
function male(p) { return p.sex == "m"; }
function female(p) { return p.sex == "f"; }

console.log(average(ancestry.filter(male).map(age)));
// → 61.67
console.log(average(ancestry.filter(female).map(age)));
// → 54.56
```

(((plus function)))(((+ operator)))(((function,as value)))(It's a bit silly that we have to define `plus` as a function, but operators in JavaScript, unlike functions, are not values, so you can't pass them as arguments.)

(((abstraction)))(((vocabulary)))Instead of tangling the logic into a big ((loop)), it is neatly composed into the concepts we are interested in—determining sex, computing age, and averaging numbers. We can apply these one by one to get the result we are looking for.

This is *fabulous* for writing clear code. Unfortunately, this clarity comes at a cost.

== The cost ==

(((efficiency)))(((optimization)))In the happy land of elegant code and pretty rainbows, there lives a spoil-sport monster called *inefficiency*.

(((elegance)))(((array,creation)))(((pure function)))(((composability)))A program that processes an array is most elegantly expressed as a sequence of cleanly separated steps that each do something with the array and produce a new array. But building up all those intermediate arrays is somewhat expensive.

(((readability)))(((function,application)))(((forEach method)))(((function,as value)))Likewise, passing a function to `forEach` and letting that method handle the array iteration for us is convenient and easy to read. But function calls in JavaScript are costly compared to simple loop bodies.

(((abstraction)))And so it goes with a lot of techniques that help improve the clarity of a program. Abstractions add layers between the raw things the computer is doing and the concepts we are working with and thus cause the machine to perform more work. This is not an iron law—there are programming languages that have better support for building abstractions without adding inefficiencies, and even in JavaScript, an experienced programmer can find ways to write abstract code that is still fast. But it is a problem that comes up a lot.

(((profiling)))Fortunately, most computers are insanely fast. If you are processing a modest set of data or doing something that has to happen only on a human time scale (say, every time the user clicks a button), then it *does not matter* whether you wrote a pretty solution that takes half a millisecond or a super-optimized solution that takes a tenth of a millisecond.

(((nesting,of loops)))(((inner loop)))(((complexity)))It is helpful to roughly keep track of how often a piece of your program is going to run. If you have a ((loop)) inside a loop (either directly or through the outer loop calling a function that ends up performing the inner loop), the code inside the inner loop will end up running **N×M** times, where *N* is the number of times the outer loop repeats and *M* is the number of times the inner loop repeats within each iteration of the outer loop. If that inner loop contains another loop that makes *P* rounds, its body will run **M×N×P** times, and so on. This can add up to large numbers, and when a program is slow, the problem can often be traced to only a small part of the code, which sits inside an inner loop.

== Great-great-great-great-... ==

(((ancestry example)))My ((grandfather)), Philibert Haverbeke, is included in the data file. By starting with him, I can trace my lineage to find out whether the most ancient person in the data, Pauwels van Haverbeke, is my direct ancestor. And if he is, I would like to know how much ((DNA)) I theoretically share with him.

(((byName object)))(((map)))(((data structure)))(((object,as map)))To be able to go from a parent's name to the actual object that represents this person, we first build up an object that associates names with people.

// include_code strip_log

```
var byName = {};
ancestry.forEach(function(person) {
  byName[person.name] = person;
});

console.log(byName["Philibert Haverbeke"]);
// → {name: "Philibert Haverbeke", …}
```

Now, the problem is not entirely as simple as following the `father` properties and counting how many we need to reach Pauwels. There are several cases in the family ((tree)) where people married their second cousins (tiny villages and all that). This causes the branches of the family tree to rejoin in a few places, which means I share more than 1/2^*G*^ of my genes with this person, where *G* for the number of generations between Pauwels and me. This formula comes from the idea that each generation splits the gene pool in two.

(((reduce method)))(((data structure)))A reasonable way to think about this problem is to look at it as being analogous to `reduce`, which condenses an array to a single value by repeatedly combining values, left to right. In this case, we also want to condense our data structure to a single value but in a way that follows family lines. The *shape* of the data is that of a family tree, rather than a flat list.

The way we want to reduce this shape is by computing a value for a given person by combining values from their ancestors. This can be done recursively: if we are interested in person *A*, we have to compute the values for **A**'s parents, which in turn requires us to compute the value for **A**'s grandparents, and so on. In principle, that'd require us to look at an infinite number of people, but since our data set is finite, we have to stop somewhere. We'll allow a ((default value)) to be given to our reduction function, which will be used for people who are not in the data. In our case, that value is simply zero, on the assumption that people not in the list don't share DNA with the ancestor we are looking at.

(((recursion)))(((reduceAncestors function)))Given a person, a function to combine values from the two parents of a given person, and a default value, `reduceAncestors` condenses a value from a family tree.

// include_code

```
function reduceAncestors(person, f, defaultValue) {
  function valueFor(person) {
    if (person == null)
      return defaultValue;
    else
      return f(person, valueFor(byName[person.mother]),
                       valueFor(byName[person.father]));
  }
  return valueFor(person);
}
```

(((function,higher-order)))The inner function (`valueFor`) handles a single person. Through the ((magic)) of recursion, it can simply call itself to handle the father and the mother of this person. The results, along with the person object itself, are passed to `f` , which returns the actual value for this person.

We can then use this to compute the amount of ((DNA)) my ((grandfather)) shared with Pauwels van Haverbeke and divide that by four.

// start_code bottom_lines: 2 // test: clip // include_code top_lines: 6

```
function sharedDNA(person, fromMother, fromFather) {
  if (person.name == "Pauwels van Haverbeke")
    return 1;
  else
    return (fromMother + fromFather) / 2;
}
var ph = byName["Philibert Haverbeke"];
console.log(reduceAncestors(ph, sharedDNA, 0) / 4);
// → 0.00049
```

The person with the name Pauwels van Haverbeke obviously shared 100 percent of his DNA with Pauwels van Haverbeke (there are no people who share names in the data set), so the function returns 1 for him. All other people share the average of the amounts that their parents share.

So, statistically speaking, I share about 0.05 percent of my ((DNA)) with this 16th-century person. It should be noted that this is only a statistical approximation, not an exact amount. It is a rather small number, but given how much genetic material we carry (about 3 billion base pairs), there's still probably some aspect in the biological machine that is me that originates with Pauwels.

(((ancestry example)))(((reduceAncestors function)))(((abstraction)))We could also have computed this number without relying on `reduceAncestors` . But separating the general approach (condensing a family tree) from the specific case (computing shared DNA) can

improve the clarity of the code and allows us to reuse the abstract part of the program for other cases. For example, the following code finds the percentage of a person's known ancestors who lived past 70 (by lineage, so people may be counted multiple times):

// test: clip

```
function countAncestors(person, test) {
  function combine(current, fromMother, fromFather) {
    var thisOneCounts = current != person && test(current);
    return fromMother + fromFather + (thisOneCounts ? 1 : 0);
  }
  return reduceAncestors(person, combine, 0);
}
function longLivingPercentage(person) {
  var all = countAncestors(person, function(person) {
    return true;
  });
  var longLiving = countAncestors(person, function(person) {
    return (person.died - person.born) >= 70;
  });
  return longLiving / all;
}
console.log(longLivingPercentage(byName["Emile Haverbeke"]));
// → 0.129
```

Such numbers are not to be taken too seriously, given that our data set contains a rather arbitrary collection of people. But the code illustrates the fact that `reduceAncestors` gives us a useful piece of ((vocabulary)) for working with the family tree data structure.

== Binding ==

(((bind method)))(((partial application)))(((function,application)))The `bind` method, which all functions have, creates a new function that will call the original function but with some of the arguments already fixed.

(((filter method)))(((function,as value)))The following code shows an example of `bind` in use. It defines a function `isInSet` that tells us whether a person is in a given set of strings. To call `filter` in order to collect those person objects whose names are in a specific set, we can either write a function expression that makes a call to `isInSet` with our set as its first argument or *partially apply* the `isInSet` function.

```
var theSet = ["Carel Haverbeke", "Maria van Brussel",
              "Donald Duck"];
function isInSet(set, person) {
  return set.indexOf(person.name) > -1;
}

console.log(ancestry.filter(function(person) {
  return isInSet(theSet, person);
}));
// → [{name: "Maria van Brussel", …},
//    {name: "Carel Haverbeke", …}]
console.log(ancestry.filter(isInSet.bind(null, theSet)));
// → … same result
```

The call to `bind` returns a function that will call `isInSet` with `theSet` as first argument, followed by any remaining arguments given to the bound function.

(((null)))The first argument, where the example passes `null`, is used for ((method call))s, similar to the first argument to `apply`. I'll describe this in more detail in the link:06_object.html#call_method[next chapter].

== Summary ==

Being able to pass function values to other functions is not just a gimmick but a deeply useful aspect of JavaScript. It allows us to write computations with “gaps” in them as functions and have the code that calls these functions fill in those gaps by providing function values that describe the missing computations.

Arrays provide a number of useful higher-order methods— `forEach` to do something with each element in an array, `filter` to build a new array with some elements filtered out, `map` to build a new array where each element has been put through a function, and `reduce` to combine all an array's elements into a single value.

Functions have an `apply` method that can be used to call them with an array specifying their arguments. They also have a `bind` method, which is used to create a partially applied version of the function.

== Exercises ==

=== Flattening ===

(((flattening (exercise)))))(((reduce method)))(((concat method)))(((array)))Use the `reduce` method in combination with the `concat` method to “flatten” an array of arrays into a single array that has all the elements of the input arrays.

ifdef::interactive_target[]

// test: no

```
var arrays = [[1, 2, 3], [4, 5], [6]];
// Your code here.
// → [1, 2, 3, 4, 5, 6]
```

endif::interactive_target[]

=== Mother-child age difference ===

(((ancestry example)))(((age difference (exercise))))(((average function)))Using the example data set from this chapter, compute the average age difference between mothers and children (the age of the mother when the child is born). You can use the `average` function defined link:05_higher_order.html#average_function[earlier] in this chapter.

(((byName object)))Note that not all the mothers mentioned in the data are themselves present in the array. The `byName` object, which makes it easy to find a person's object from their name, might be useful here.

ifdef::interactive_target[]

// test: no // include_code

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

var byName = {};
ancestry.forEach(function(person) {
  byName[person.name] = person;
});

// Your code here.

// → 31.2
```

endif::interactive_target[]

!!hint!!

(((age difference (exercise))))(((filter method)))(((map method)))(((null)))(((average function)))Because not all elements in the `ancestry` array produce useful data (we can't compute the age difference unless we know the birth date of the mother), we will have to apply `filter` in some manner before calling `average`. You could do it as a first pass, by defining a `hasKnownMother` function and filtering on that first. Alternatively, you could start by

calling `map` and in your mapping function return either the age difference or `null` if no mother is known. Then, you can call `filter` to remove the `null` elements before passing the array to `average`.

!!hint!!

=== Historical life expectancy ===

(((life expectancy (exercise))))When we looked up all the people in our data set that lived more than 90 years, only the latest generation in the data came out. Let's take a closer look at that phenomenon.

(((average function)))Compute and output the average age of the people in the ancestry data set per century. A person is assigned to a ((century)) by taking their year of death, dividing it by 100, and rounding it up, as in `Math.ceil(person.died / 100)`.

ifdef::interactive_target[]

// test: no

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

// Your code here.

// → 16: 43.5
//   17: 51.2
//   18: 52.8
//   19: 54.8
//   20: 84.7
//   21: 94
```

endif::interactive_target[]

!!hint!!

(((life expectancy (exercise))))The essence of this example lies in ((grouping)) the elements of a collection by some aspect of theirs—splitting the array of ancestors into smaller arrays with the ancestors for each century.

(((array)))(((map)))(((object,as map)))During the grouping process, keep an object that associates ((century)) names (numbers) with arrays of either person objects or ages. Since we do not know in advance what categories we will find, we'll have to create them on the fly.

For each person, after computing their century, we test whether that century was already known. If not, add an array for it. Then add the person (or age) to the array for the proper century.

(((for/in loop)))(((average function)))Finally, a `for` / `in` loop can be used to print the average ages for the individual centuries.

!!hint!!

(((grouping)))(((map)))(((object,as map)))(((groupBy function)))For bonus points, write a function `groupBy` that abstracts the grouping operation. It should accept as arguments an array and a function that computes the group for an element in the array and returns an object that maps group names to arrays of group members.

=== Every and then some ===

(((predicate function)))(((every and some (exercise))))(((every method)))(((some method)))(((array,methods)))(((&& operator)))(((|| operator)))Arrays also come with the standard methods `every` and `some`. Both take a predicate function that, when called with an array element as argument, returns true or false. Just like `&&` returns a true value only when the expressions on both sides are true, `every` returns true only when the predicate returns true for *all* elements of the array. Similarly, `some` returns true as soon as the predicate returns true for *any* of the elements. They do not process more elements than necessary—for example, if `some` finds that the predicate holds for the first element of the array, it will not look at the values after that.

Write two functions, `every` and `some`, that behave like these methods, except that they take the array as their first argument rather than being a method.

ifdef::interactive_target[]

// test: no

```
// Your code here.

console.log(every([NaN, NaN, NaN], isNaN));
// → true
console.log(every([NaN, NaN, 4], isNaN));
// → false
console.log(some([NaN, 3, 4], isNaN));
// → true
console.log(some([2, 3, 4], isNaN));
// → false
```

endif::interactive_target[]

!!hint!!

(((every and some (exercise))))(((short-circuit evaluation)))(((return keyword)))The functions can follow a similar pattern to the link:05_higher_order.html#forEach[definition] of `forEach` at the start of the chapter, except that they must return immediately (with the right value) when the predicate function returns false—or true. Don't forget to put another `return` statement after the loop so that the function also returns the correct value when it reaches the end of the array.

!!hint!!

La Vida Secreta de los Objetos

> El problema con los lenguajes orientados a objetos es que tienen todos un medio implícito que llevan consigo. Tu quieres un plátano pero eres un gorila con un plátano y la jungla entera. Joe Armstrong, entrevistado en Coders at Work

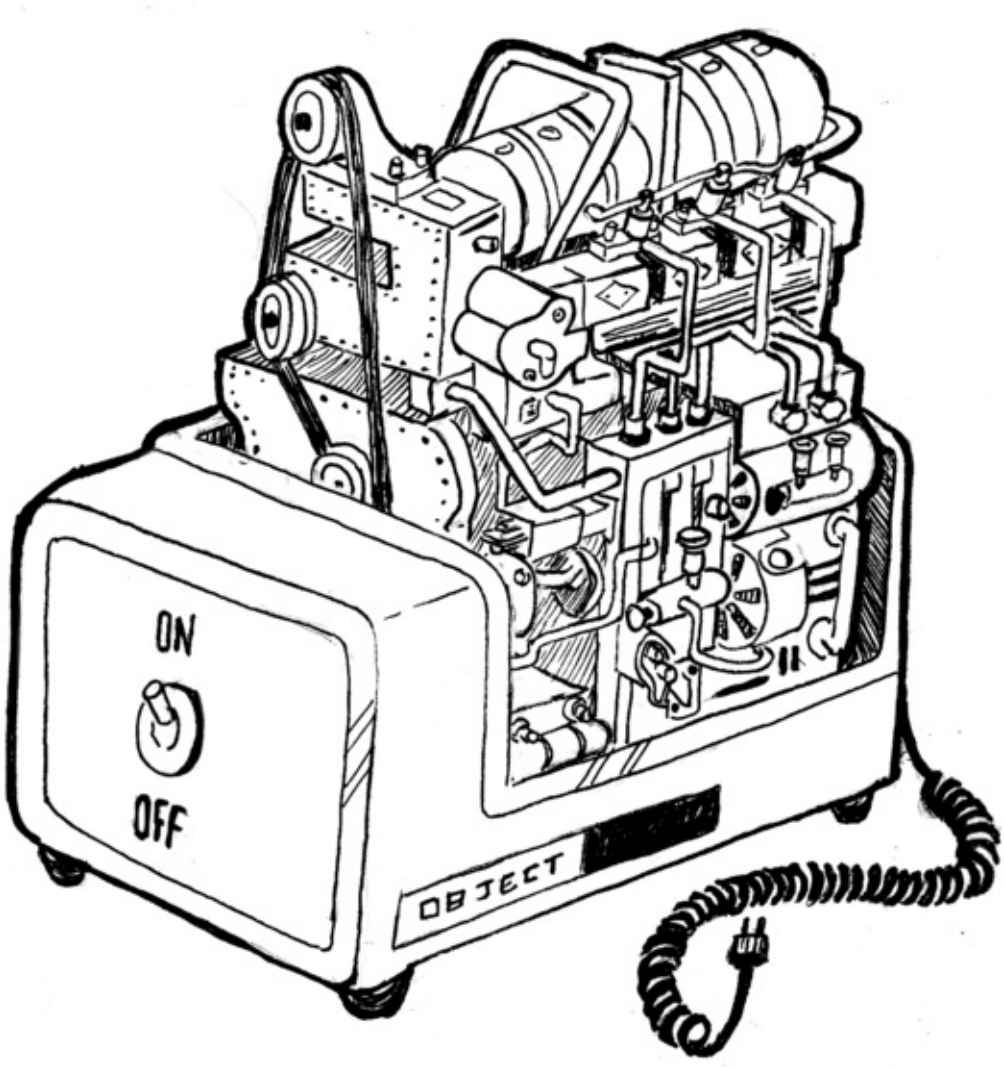
Cuando un programador dice ″objeto″, este es un término amplio. En mi profesión, los objetos son una forma de vida, tema de guerras sagradas y una amada palabra llamativa que todavía no ha perdido su gran poder.

Para alguien ajeno, esto es probablemente un poco confuso. Empecemos con una introducción a la *historia* de los objetos como una forma de programación.

Historia

Esta historia, como la mayor parte de las historias de programación, comienza con el problema de la *complejidad*. Una filosofía es que la complejidad puede ser manejable separándola en pequeñas partes, que son aisladas unas de otras. Estas partes han terminado con el nombre de *objetos*.

Un objeto es una cápsula opaca que oculta una sofisticada complejidad en su interior y en su lugar nos ofrece unos pocos reguladores y conectores (como por ejemplo *método_s) que presentan una _interfaz* a través de la cual el objeto es usado. La idea es que la la interfaz es relativamente simple y todas las cosas complejas que van *dentro* del objeto pueden ser ignoradas cuando trabajamos con el.



Como ejemplo, puedes imaginarte un objeto que te dé una interfaz para un área de la pantalla. Te da una forma de dibujar formas o texto en esta área pero oculta los detalles de como esas formas son convertidas a los pixeles que decoran la pantalla actualmete. Puedes tener un conjunto de métodos-por ejemplo `dibujarCirculo` -y estos son lo único que necesitas para usar un objeto.

Estas ideas fueron puestas en marcha en los 70, los 80 y los 90, fueron acompañadas por un gran *bombo*-la revolución de la programación orientada a objetos. Inmediatamente había un grupo de gente declarando que los objetos eran el camino *correcto* a la programación-y que no incluir objetos era un sin sentido, estaba obsoleto.

Este tipo de fanatismo siempre produce mucha estupidez no práctica y ha habido una pequeña contra revolución después de esto. Actualmente en algunos círculos, los objetos tienen una reputación bastante mala.

Yo prefiero abordar el tema desde la práctica, en lugar de desde la ideología. Hay varios conceptos útiles, más importantes que la *encapsulación* (distinguir entre la complejidad interna y externa de la interfaz), que la cultura de la programación orientada a objetos a

popularizado. Estos son dignos de estudio.

En este capítulo se describen de forma excéntrica los objetos y las técnicas clásicas sobre como se relacionan entre sí los objetos en JavaScript.

Métodos

Los métodos son propiedades simples que contienen funciones como valores. Este es un método simple:

```
var conejo = {};
conejo.hablar = function(linea) {
  console.log("El conejo dice '" + linea + "'");
};

conejo.hablar("Estoy vivo.");
// → El conejo dice 'Estoy vivo.'
```

Normalmente el método necesita hacer algo con el objeto desde el que se le ha llamado. Cuando una función es llamada como método-se busca como propiedad y es inmediatamente llamada, como en `objeto.metodo()` —la variable especial `this` esta en su cuerpo y apuntará al objeto que la ha llamado.

```
function hablar(linea) {
  console.log("El conejo " + this.tipo + " dice '" +
              line + "'");
}
var conejoBlanco = {tipo: "blanco", hablar: hablar};
var conejoGordo = {tipo: "gordo", hablar: hablar};

conejoBlanco.hablar("¡Por mis orejas y los pelos de mi " +
                    "bigote, que tarde se está haciendo!");
// → El conejo blanco dice '¡Por mis orejas y los pelos'
//   de mi bigote, que tarde se está haciendo!'
conejoGordo.hablar("Puedes estar seguro de que me comería +"
                    "una zanahoria.");
// → El conejo gordo dice 'Puedes estar seguro de que
//   me comería una zanahoria.'
```

El código usa la palabra clave `this` para la salida del tipo de conejo que está hablando. Se puede rellamar con los métodos `apply` y `bind`, ambos toman un primer argumento que puede ser utilizado para simular llamadas al método. El primer argumento es de echo utilizado para dar valor a `this`.

Hay un método similar a `apply`, llamado `call`. Este llama a la función que es un método, pero toma sus argumentos normalmente, en lugar de con un array. Como `apply` y `bind`, `call` puede pasar un valor específico de `this`.

```
hablar.apply(conejoGordo, ["¡Burp!"]);
// → El conejo gordo dice '¡Burp!'
hablar.call({tipo: "viejo"}, "¡Oh!, ¡Ah!");
// → El conejo viejo dice '¡Oh!, ¡Ah!'
```

Prototipos

(Fíjate detenidamente).

```
var vacio = {};
console.log(vacio.toString);
// → function toString(){…}
console.log(vacio.toString());
// → [object Object]
```

Acabo de extraer una propiedad de un objeto vacío. ¡Magia!

Bien, no realmente. Simplemente he omitido información acerca de como los Objetos funcionan en JavaScript. Además de sus propiedades, casi todos los objetos además tienen un *prototipo*. Un *prototipo* es otro objeto que es usado como alternativa fuente de propiedades. Cuando un objeto tiene una llamada a una propiedad que no posee, se buscará en su prototipo, después en el prototipo de su prototipo y así sucesivamente.

Entonces, ¿Cual es el *prototipo* de este objeto vacío? Es el genial prototipo ancestral, la entidad detrás de casi todos los objetos, `Object.prototype`.

```
console.log(Object.getPrototypeOf({}) ==
            Object.prototype);
// → true
console.log(Object.getPrototypeOf(Object.prototype));
// → null
```

Como imaginarás la función `Object.getPrototypeOf` devuelve el prototipo de un objeto.

Las relaciones de prototipo en JavaScript tienen forma de *árbol*, y la raíz de esta estructura es `Object.prototype`. Este provee unos pocos *métodos* que se mostrarán en casi todos los objetos, como `toString`, que convierte un objeto en una representación en una cadena de texto.

Muchos objetos no tienen directamente `Object.prototype` como su *prototipo*, pero en su lugar tienen otro objeto, que les provee sus propiedades por defecto. Las funciones derivan de `Function.prototype`, y los arrays derivan de `Array.prototype`.

```
console.log(Object.getPrototypeOf(isNaN) ==
            Function.prototype);
// → true
console.log(Object.getPrototypeOf([]) ==
            Array.prototype);
// → true
```

Como un objeto prototipo tiene su propio prototipo normalmente `Object.prototype`, entonces este indirectamente provee de métodos como `toString`.

La función `Object.getPrototypeOf` obviamente devuelve el prototipo de un objeto. Puedes usar `Object.create` para crear un objeto con un *prototipo* específico.

```
var protoConejo = {
  hablar: function(linea) {
    console.log("El conejo " + this.type + " dice '" +
                linea + "'");
  }
};
var conejoAsesino = Object.create(protoConejo);
conejoAsesino.type = "asesino";
conejoAsesino.hablar("SKREEEE!");
// → El conejo asesino dice 'SKREEEE!'
```

El “proto” conejo actúa como container para las propiedades que son compartidas por todos los conejos. Un objeto conejo individual, como el conejo asesino, contiene propiedades que se aplican únicamente a sí mismo, en este caso su tipo y propiedades derivadas de su prototipo.

Constructores

Una forma más conveniente de crear objetos que deriven su forma de prototipos compartidos es usar un *constructor*. En JavaScript, llamar a una función con la palabra clave `new` delante de ella, hace que sea tratada como un constructor. El constructor tendrá su variable `this` en los límites del objeto creado, y si no se específica otro valor de objeto este será el nuevo objeto que retorne la llamada.

Un objeto creado con `new` se dice que es una *instancia* de su constructor.

Tenemos un constructor simple para los conejos. Es una convención capitalizar (poner la primera letra en mayúscula) los nombres de los constructores así son fácilmente distinguidos de otras funciones.

```
function Conejo(tipo) {
  this.tipo = tipo;
}

var conejoAsesino = new Conejo("asesino");
var conejoNegro = new Conejo("negro");
console.log(conejoNegro.tipo);
// → negro
```

Los constructores (de hecho, todas las funciones) automáticamente tienen una propiedad llamada `prototype`, que por defecto contiene un objeto plano, vacío que deriva de `Object.prototype`. Todas las instancias creadas con este constructor tendrán este objeto como su ((prototipo)). Así que para añadir un método `hablar` a los conejos creados con el constructor `Conejo`, simplemente hacemos lo siguiente:

```
Conejo.prototype.hablar = function(linea) {
  console.log("El conejo " + this.tipo + " dice '" +
              linea + "'");
};
conejoNegro.hablar("Maldición...");
// → El conejo negro dice 'Maldición...'
```

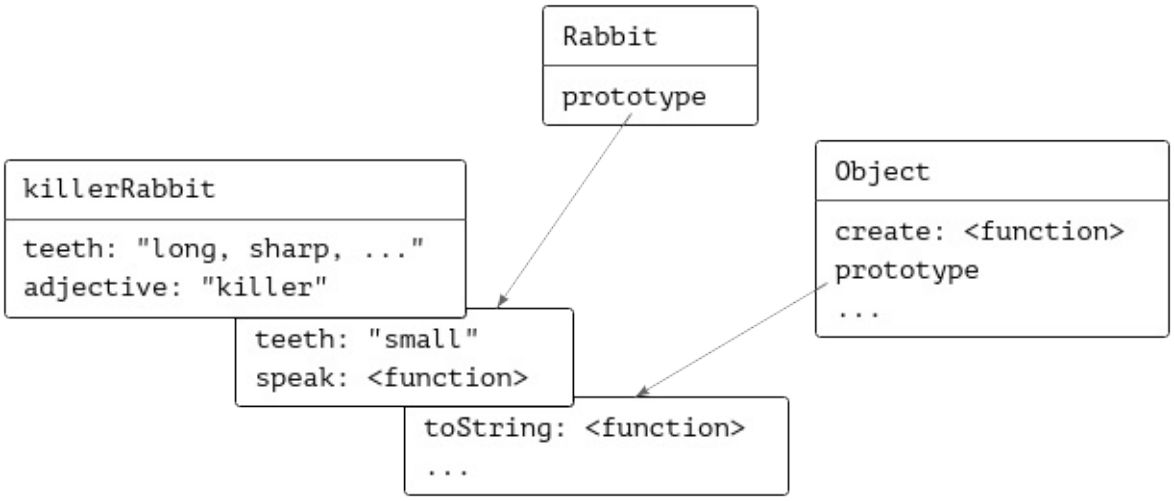
Es importante notar la diferencia entre la forma en que un prototipo es asociado con un constructor (a través de su propiedad `prototype`) y la forma en la que los objetos *tienen* un prototipo (que podemos consultar con `Object.getPrototypeOf`). El prototipo actual de un constructor es `Function.prototype` desde que los constructores son funciones. Esta propiedad `prototype` será el prototipo de las instancias creadas a través de el pero no su *propio* prototipo.

Sobre Escribiendo Las Propiedades Derivadas

Cuando añades una *propiedad* a un objeto, esté presente en el prototipo o no, la propiedad es añadida a *ese* objeto, que de ahora en adelante tendrá como su propiedad. Si *existe* una propiedad con el mismo nombre en el prototipo, esta propiedad no afectará más al objeto. El prototipo por si mismo no cambia.

```
Conejo.prototype.dentadura = "pequeña";
console.log(conejoNegro.dentadura);
// → pequeña
conejoAsensino.dentadura = "larga, afilada y sangrienta";
console.log(conejoAsesino.dentadura);
// → larga, afilada y sangrienta
console.log(conejoNegro.dentadura);
// → pequeña
console.log(Conejo.prototype.dentadura);
// → pequeña
```

El siguiente diagrama representa la situación después de ejecutar este código. El `Conejo` y `Objeto` _prototipo_s están detrás de `conejoAsesino` como una especie telón de fondo, donde sus propiedades que no son encontradas en el objeto por si mismo pueden ser buscadas.



Sobre escribir propiedades que existen en un prototipo, es a menudo algo útil que hacer. Como muestra el ejemplo de la dentadura del conejo, esto puede ser usado para expresar propiedades excepcionales en instancias de una clase más genérica de objetos, mientras dejamos los objetos no excepcionales simplemente tomar un valor estándar de su prototipo.

Esto es además usado para dar a los prototipos de función y array un método `toString` diferente del básico prototipo de los objetos.

```
console.log(Array.prototype.toString ==
            Object.prototype.toString);
// → false
console.log([1, 2].toString());
// → 1,2
```

Llamar a `toString` en un array da un resultado similar a `.join(",")` -esto pone comas entre los valores del array. Una llamada directa a `Object.prototype.toString` con un array produce una cadena de texto diferente. Esta función no sabe acerca de arrays, así que

simplemente pone la palabra "object" y el nombre del tipo entre corchetes.

```
console.log(Object.prototype.toString.call([1, 2]));
// → [object Array]
```

Interferencia de prototipos

Un *prototipo* puede ser usado en cualquier momento para añadir nuevas propiedades y métodos a todos los objetos basados en él. Por ejemplo, puede ser necesario para poner a nuestros conejos a bailar.

```
Conejo.prototype.bailar = function() {
  console.log("El conejo " + this.type + " baila un paso.");
};
conejoAsesino.bailar();
// → El conejo asesino baila un paso.
```

Esto es conveniente. Pero hay situaciones donde esto causa problemas. En capítulos anteriores, hemos usado un objeto como forma de asociar valores con nombres creando propiedades para los nombres y dándoles los correspondientes valores como su valor. Aquí hay un ejemplo Capitulo 4:

```
var mapa = {};
function guardarPhi(evento, phi) {
  mapa[evento] = phi;
}

guardarPhi("pizza", 0.069);
guardarPhi("árbol tocado", -0.081);
```

Podemos iterar sobre todos los valores de phi en el objeto usando un bucle `for` / `in` y comprobar cuando un nombre esta usando el operador regular `in`. Pero desafortunadamente, el objeto del prototipo continua con su camino.

```
Object.prototype.sinSentido = "hola";
for (var nombre in mapa)
  console.log(nombre);
// → pizza
// → árbol tocado
// → sinSentido
console.log("sinSentido" in mapa);
// → true
console.log("toString" in mapa);
// → true

// Borrar la propiedad problemática
delete Object.prototype.sinSentido;
```

Esta todo mal. No hay evento llamado "sinSentido" en nuestro set de datos. Y *definitivamente* no hay evento llamado "toString".

Extrañamente, `toString` no se muestra en el bucle `for` / `in` , pero el operador `in` ha retornado true para el. Esto es por que JavaScript distingue entre propiedades *enumerable* (enumerables) y *nonenumerable* (no enumerables).

Todas las propiedades que creamos simplemente asignándolas son enumerables. Las propiedades estándar en `Object.prototype` son todas nonenumerable, que es por lo que no se muestran en un bucle como un `for` / `in` .

Es posible definir nuestras propias propiedades nonenumberable usando la función `Object.defineProperty` , esta nos permite controlar el tipo de propiedad que estamos creando.

```
Object.defineProperty(Object.prototype, "ocultarSinSentido",
                      {enumerable: false, value: "hola"});
for (var nombre in mapa)
  console.log(nombre);
// → pizza
// → árbol tocado
console.log(mapa.ocultarSinSentido);
// → hola
```

Entonces ahora la propiedad esta, pero no se muestra en un bucle. Esto es bueno. Pero seguimos teniendo el problema con el operador regular `in` demandando que las propiedades del `Object.prototype` existen en nuestro objeto. Para esto, podemos usar el método de objeto `hasOwnProperty` .

```
console.log(mapa.hasOwnProperty("toString"));
// → false
```

Este método nos dice cuando el objeto *por si mismo* tiene la propiedad, sin mirar en sus prototipos. Esto es a menudo una información más útil que la que nos da el operador `in`.

Cuando tu estás preocupado de que algo (algún otro código que has incluido en tu programa) puede tener problemas con el objeto base prototipo, te recomiendo escribir bucles `for` / `in` como este:

```
for (var nombre in mapa) {
  if (mapa.hasOwnProperty(nombre)) {
    // ... esta es una propiedad propia
  }
}
```

Objetos sin prototipo

Pero el agujero del conejo no acaba aquí. ¿Qué pasa si alguien registra el nombre `hasOwnProperty` en nuestro objeto `mapa` y le asigna el valor 42? Ahora la llamada a `mapa.hasOwnProperty` intentará llamar a la propiedad local, que contiene un número, no una función.

En este caso, los prototipos solo continúan su camino y nosotros podemos preferir tener objetos sin prototipos por ahora. Vemos la función `Object.create`, que nos permite crear un objeto con un prototipo específico. Le puedes pasar `null` como prototipo para crear un objeto vacío sin prototipo. Para objetos como `map`, donde las propiedades pueden ser cualquiera, esto es exactamente lo que queremos.

```
var mapa = Object.create(null);
mapa["pizza"] = 0.069;
console.log("toString" in mapa);
// → false
console.log("pizza" in mapa);
// → true
```

¡Mucho mejor! Ya no necesitaremos la chapuza de `hasOwnProperty` por que todas las propiedades que el objeto tiene son sus propias propiedades. Ahora podemos usar de forma segura bucles `for` / `in`, no hay problema con lo que la gente le haya estado haciendo a `Object.prototype`.

Polimorfismo

Cuando llamas a la función `String`, que convierte un valor en una cadena, en un objeto, esta llamará al método `toString` cuando el objeto trate de crear una cadena con sentido para retornarla. He mencionado que alguno de los prototipos estándar definen su propia

versión de `toString` así que ellos pueden crear cadenas que contengan información más útil que `"[object Object]"`.

Esta es una simple instancia de una poderosa idea. Cuando un trozo de código es escrito para trabajar con objetos que tienen una *interfaz* concreta -en este caso, un método `toString`- entonces cualquier tipo de objeto que soporte esta interfaz y pueda ser introducido en el código, simplemente funcionará.

Esta técnica es llamada **polimorfismo**-aunque no hay cambio de forma real actualmente involucrado. El código polimórfico puede trabajar con valores de diferentes formas, tantas como sean soportadas por la interfaz.

Dando estilo a una tabla

Voy a trabajar a través de un ejemplo un poco más complicado en un intento de darte una idea mejor de como se utiliza el *polimorfismo* y la *programación orientada a objetos* en general. El proyecto es este: escribiremos un programa que, dado un array de arrays, de *tabla* celdas, construya una cadena de texto que contenga un genial diseño de tabla-significa que las columnas y las filas están correctamente alineadas. Algo como esto:

```
nombre       altura país
------------ ------ -------------
Kilimanjaro    5895 Tanzania
Everest        8848 Nepal
Mount Fuji     3776 Japan
Mont Blanc     4808 Italy/France
Vaalserberg     323 Netherlands
Denali         6168 United States
Popocatepetl   5465 Mexico
```

La forma en que nuestro sistema de generación de tablas funcionará es que la función generadora preguntará a cada celda cual va a ser su ancho y alto y después usar esa información para determinar la anchura de las columnas y la altura de las filas. La función generadora después pedirá a las celdas que se dibujen a sí mismas con el tamaño correcto y ensamblando los resultados en una sola cadena.

El programa de estilo se comunicará con los objetos celda a través de una interfaz bien definida. De esta forma, los tipos de celda que el programa soporta no estarán fijados. Podremos añadir nuevos tipos de celda más adelante-por ejemplo, celdas subrayadas para la cabecera de la tabla-y si lo soporta nuestra interfaz, simplemente funcionará, sin requerir cambios al programa de diseño.

Esta es la interfaz:

- `minAltura()` devuelve un número indicando la altura mínima que la celda requiere (en lineas).
- `minAnchura()` devuelve un número indicando la anchura mínima de esta celda en caracteres).
- `dibujar(anchura, altura)` devuelve un array de tamaño `altura`, que contiene una serie de cadenas que son cada `anchura` en caracteres. Esto representa el contenido de la celda.

Voy a hacer uso intensivo de métodos de orden superior en arrays en este ejemplo, ya que se presta bien a este enfoque.

La primera parte del programa calcula arrays de los mínimos anchos de columna y altos de fila para una grilla de celdas. La variable `filas` contendrá un array de arrays, con cada array interno representado una fila de celdas.

```
function alturasFila(filas) {
  return filas.map(function(fila) {
    return fila.reduce(function(max, celda) {
      return Math.max(max, celda.minAltura());
    }, 0);
  });
}

function anchurasColumna(filas) {
  return filas[0].map(function(_, i) {
    return filas.reduce(function(max, fila) {
      return Math.max(max, fila[i].minAnchura());
    }, 0);
  });
}
```

Usar un nombre de variable que comience con un guión bajo (_) o que consista en un simple guión bajo es una forma de indicar (a los lectores humanos) que este argumento no se utilizará.

La función `alturasFila` no debería ser demasiado difícil de seguir. Esta usa `reduce` para calcular la altura máxima de un array de celdas y está dentro de un `map` para conseguir que se haga para todas las filas en el array `filas`.

Las cosas son un poco mas complicadas para la función `anchurasColumna` por que el array exterior es un array de filas, no de columnas. Se me ha olvidado mencionar que a `map` (como a `forEach`, `filter`, y métodos similares de array) se les puede pasar un segundo argumento, este es en la función el *índice* del elemento actual. Mapeando los elementos de la primera fila y usando solo el segundo argumento de la función mapping, `colWidths`

genera un array con un elemento para cada índice de columna. La llamada a `reduce` se ejecuta sobre el array externo `filas` para cada índice y se extrae la anchura de la celda más ancha para ese índice.

Aquí esta el código para dibujar una tabla:

```
function dibujarTabla(filas) {
  var alturas = alturasFilas(filas);
  var anchuras = anchurasColumnas(filas);

  function dibujarLinea(bloques, numLinea) {
    return bloques.map(function(bloque) {
      return bloque[numLinea];
    }).join(" ");
  }

  function dibujarFila(fila, numFila) {
    var bloques = fila.map(function(celda, numColumna) {
      return celda.dibujar(anchuras[numColumna], alturas[numFila]);
    });
    return bloques[0].map(function(_, numLinea) {
      return dibujarLinea(bloques, numLinea);
    }).join("\n");
  }

  return filas.map(dibujarFila).join("\n");
}
```

La función `dibujarTabla` usa la función auxiliar interna `dibujarFila` para dibujar todas las filas y después unirlas todas con el caracteres de nueva línea.

La función `dibujarFila` por si misma convierte los objetos celda en la fila a *bloques*, que son los arrays de cadenas representando el contenido de las celdas, separados por línea. Una celda simple contiene simplemente el número 3776 puede ser representado como un elemento simple de array como `["3776"]`, como una celda subrayada nos va a ocupar dos lineas será representada por el array `["nombre", "------"]`.

Los bloques para una fila, que tienen la misma altura, deben aparecer uno junto a otro en la salida final. La segunda llamada a `map` en `dibujarFila` genera esta salida línea a línea mapeando a través de las líneas desde el bloque más a la izquierda y, para cada uno de estos, coleccionando una línea que ocupa la anchura total de la tabla. Estas líneas están unidas con el carácter nueva línea para proveer la fila entera como valor de retorno de `dibujarFila`.

La función `dibujarLinea` extrae líneas que deben aparecer unas junto a otras de un array de bloques y las une con un carácter espacio para crear un hueco de un carácter entre las columnas de la tabla.

Ahora vamos a escribir un constructor, para las celdas que contienen texto, que implementa la ((interfaz)) para las celdas de la tabla. El constructor separa una cadena en un array de líneas usando el método de string `split`, que separa una cadena en cada ocurrencia de su argumento y retorna un array de piezas. El método `minAnchura` encuentra la máxima anchura de línea en este array.

```
function repetir(cadena, veces) {
  var resultado = "";
  for (var i = 0; i < veces; i++)
    resultado += cadena;
  return resultado;
}

function CeldaTexto(texto) {
  this.texto = texto.split("\n");
}
CeldaTexto.prototype.minAnchura = function() {
  return this.texto.reduce(function(anchura, linea) {
    return Math.max(anchura, linea.length);
  }, 0);
};
CeldaTexto.prototype.minAltura = function() {
  return this.texto.length;
};
CeldaTexto.prototype.dibujar = function(anchura, altura) {
  var resultado = [];
  for (var i = 0; i < altura; i++) {
    var linea = this.texto[i] || "";
    resultado.push(linea + repetir(" ", anchura - linea.length));
  }
  return resultado;
};
```

El código usa una función auxiliar llamada `repetir` que genera una cadena cuyo valor es el argumento `cadena` repetido las `veces` que se indica. El método `dibujar` se usa para añadir “espacio“ a las líneas ya que todas ellas tiene la longitud requerida.

Vamos a probar lo que hemos escrito hasta ahora generando un damero de 5 x 5.

```
var filas = [];
for (var i = 0; i < 5; i++) {
   var fila = [];
   for (var j = 0; j < 5; j++) {
     if ((j + i) % 2 == 0)
       fila.push(new CeldaTexto("##"));
     else
       fila.push(new CeldaTexto("  "));
   }
   filas.push(fila);
}
console.log(dibujarTabla(filas));
// → ##    ##    ##
//      ##    ##
//   ##    ##    ##
//      ##    ##
//   ##    ##    ##
```

¡Esto funciona! Pero como todas las celda tienen la misma anchura, el código de diseñar tabla no hace algo realmente interesante.

La fuente de datos de la tabla de las montañas que estamos tratando de generar esta disponible en la variable `MOUNTAINS` en el ((sandbox)) y además es http://eloquentjavascript.net/code/mountains.js[descargable] y desde la web (!book (http://eloquentjavascript.net/code#6[_eloquentjavascript.net/code#6_])!).

Queremos destacar la fila de arriba, que contiene los nombres de las columnas, subrayando las celdas con una serie de caracteres guión. No hay problema-simplemente escribiremos un tipo de celda que soporte subrayado.

```
function CeldaSubrayada(contenido) {
  this.contenido = contenido;
};
CeldaSubrayada.prototype.minAnchura = function() {
  return this.contenido.minAnchura();

function UnderlinedCell(inner) {
  this.inner = inner;
}
UnderlinedCell.prototype.minWidth = function() {
  return this.inner.minWidth();
};
CeldaSubrayada.prototype.minAltura = function() {
  return this.contenido.minAltura() + 1;
};
CeldaSubrayada.prototype.dibujar = function(anchura, altura) {
  return this.contenido.dibujar(anchura, altura - 1)
    .concat([repetir("-", anchura)]);
};
```

Una celda subrayada contiene otra celda. Esto significa que su tamaño mínimo será el mismo que el de la celda interna (llamando a través de los métodos de estas celdas `minAnchura` y `minAltura`) pero añade uno a la altura para contar el espacio tomado por el subrayado.

Dibujar una celda es muy simple -nosotros tomamos el contenido de la celda interior y le concatenamos a una línea simple de guiones.

Teniendo un mecanismo de subrayado, ahora podemos escribir una función que genere una grilla de celdas para nuestro set de datos.

```
function datosTabla(datos) {
  var keys = Object.keys(datos[0]);
  var encabezados = keys.map(function(nombre) {
    return new CeldaSubrayada(new TextCell(nombre));
  });
  var cuerpo = datos.map(function(row) {
    return keys.map(function(nombre) {
      return new CeldaTexto(String(row[nombre]));
    });
  });
  return [encabezados].concat(cuerpo);
}

console.log(dibujarTabla(datosTabla(MOUNTAINS)));
// → nombre       altura país
//   ------------ ------ -------------
//   Kilimanjaro  5895   Tanzania
//   … etcétera
```

La función estándar `Object.keys` retorna un array de nombres de propiedades en un objeto. La fila de arriba de la tabla debe contener celdas subrayadas que den los nombres a las columnas. Debajo, los valores de todos los objetos en el set de datos parecen celdas normales-los extraeremos mapeando sobre el array `keys` así que estamos seguros de que el orden de las celdas es el mismo en cada fila.

La tabla resultante parece la del ejemplo mostrado antes, excepto por que on tiene el alineamiento a la derecha de los número en la columna `altura`. Vamos a conseguirlo en un momento.

Getters y setters

Cuando especificamos una interfaz, es posible incluir propiedades que no son métodos. Podemos tener definida `minAltura` y `minAnchura` para simplemente almacenar números. Pero esto podría requerir que lo calculáramos en él ((constructor)), esto añade código en el que no es estrictamente relevante para *construir* el objeto. Esto podría causar problemas si, por ejemplo, el interior de una celda subrayada cambia, en este punto el tamaño del subrayado de la celda debería cambiar también.

Esto ha servido como excusa para adoptar el principio de no incluir nunca propiedades que no sean métodos en las interfaces. Más que un acceso directo a un propiedad de valor simple, se pueden usar los métodos `getAlgo` y `setAlgo` para leer y escribir la propiedad. Esta aproximación tiene el inconveniente de que tu tienes que escribir -y leer- un montón de métodos adicionales.

Afortunadamente, JavaScript provee de una técnica que nos da lo mejor de ambos mundos. Podemos especificar propiedades que, dese fuera, parezcan propiedades normales pero secretamente tienen _método_s asociados con ellas.

```
var pila = {
  elementos: ["cascara de huevo", "peladura de naranja", "gusano"],
  get altura() {
    return this.elementos.length;
  },
  set altura(valor) {
    console.log("Ignorando el intento de guardar la altura: ", valor);
  }
};

console.log(pila.altura);
// → 3
pila.altura = 100;
// → Ignorando el intento de guardar la altura: 100
```

En un objeto literal, la notación `get` o `set` para propiedades te permite especificar una función para ser ejecutada cuando la propiedad es leída o escrita. Podemos incluso añadir una propiedad a un objeto existente, por ejemplo un prototipo, usando la función `Object.defineProperty` (que hemos usado previamente para crear propiedades nonenumerable).

```
Object.defineProperty(CeldaTexto.prototype, "alturaProp", {
  get: function() { return this.texto.length; }
});

var celda = new CeldaTexto("sin\nsalida");
console.log(celda.alturaProp);
// → 2
celda.alturaProp = 100;
console.log(celda.alturaProp);
// → 2
```

Puedes usar la propiedad similar `set`, en el objeto pasándola a `defineProperty`, para especificar un método setter. Cuando se define un getter pero no un setter, escribir la propiedad es simplemente ignorado.

Herencia

Todavía no hemos acabado el ejercicio de diseño de tabla. Ayuda a la legibilidad alinear a la derecha las columnas con números. Debemos crear otro tipo de celda que es como `CeldaTexto`, pero sin espacio en la parte derecha, estas tienen el espacio en la parte izquierda así que aliniémoslas a la derecha.

Podemos simplemente escribir un nuevo *constructor* entero con los tres métodos en su prototipo. Pero los prototipos pueden tener sus prototipos, y esto nos permite hacer algo inteligente.

```
function DCeldaTexto(texto) {
  CeldaTexto.call(this, texto);
}
DCeldaTexto.prototype = Object.create(CeldaTexto.prototype);
DCeldaTexto.prototype.dibujar = function(anchura, altura) {
  var resultado = [];
  for (var i = 0; i < altura; i++) {
    var linea = this.text[i] || "";
    resultado.push(repetir(" ", anchura - linea.length) + linea);
  }
  return resultado;
};
```

Reutilizamos el constructor y los métodos `minAltura` y `minAnchura` de `CeldaTexto`. Una `DCeldaTexto` es ahora básicamente equivalente a `CeldaTexto`, excepto por que su método `dibujar` contiene una función diferente.

Este patrón es llamado *herencia*. Este nos permite generar tipos de datos muy similares desde tipos de datos existente con poco trabajo relativamente. Típicamente, el nuevo constructor llamará al viejo *constructor* (usando el método `call` para permitir darle al nuevo objeto su valor `this`). Una vez este constructor se ha llamado, podemos asumir que todos los campos que el tipo de objeto viejo tenía han sido añadidos. Arreglamos el constructor del *prototipo* para derivarlo al del viejo prototipo así que las instancias de este prototipo tendrán también acceso a las propiedades del viejo prototipo. Finalmente podemos sobre escribir alguna de esas propiedades añadiéndolas a nuestro nuevo prototipo.

Ahora, si ajustamos un poco la función `datosTabla` para usar _DCeldaTexto_s para celdas cuyo valor sea un número, tendremos la tabla que estábamos buscando.

```
function datosTabla(datos) {
  var keys = Object.keys(datos[0]);
  var encabezados = keys.map(function(nombre) {
    return new CeldaSubrayada(new CeldaTexto(nombre));
  });
  var cuerpo = datos.map(function(row) {
    return keys.map(function(nombre) {
      var valor = row[nombre];
      // Esto ha cambiado:
      if (typeof valor == "number")
        return new DCeldaTexto(String(valor));
      else
        return new CeldaTexto(String(valor));
    });
  });
  return [encabezados].concat(cuerpo);
}

console.log(dibujarTabla(datosTabla(MOUNTAINS)));
// → … preciosa tabla alineada
```

La herencia es una parte fundamental de la tradición de la orientación a objetos, junto con la encapsulación y el polimorfismo. Pero mientras las dos últimas son generalmente consideradas como ideas geniales, la herencia es algo controvertido.

La principal razón para esto es que a menudo es confundida con el *polimorfismo*, vendido como una herramienta más poderosa de lo que en realidad es, y posteriormente sobre utilizado de todas las malas formas posibles. Mientras que la *encapsulación* y el

polimorfismo pueden ser usados para *separar* trozos de código de otros, reduciendo el enmarañado general del programa, la ((herencia)) fundamentalmente empata con ambos, creando *más* enmarañado.

Puedes tener polimorfismo sin herencia, como hemos visto. No te voy a decir que evites la herencia por completo-Yo la uso regularmente en mis programas. Pero tu debes verla como un truco un poco sucio que te puede ayudar a definir nuevos tipos con poco código, no como un gran principio de organización de código. Una forma mejor de extender tipos es a través de *composición*, como `CeldaSubrayada` genera otro objeto celda simplemente guardándolo en una propiedad y remitiendo las llamadas a los métodos a sus propios _métodos_s.

El operador instanceof

Es ocasionalmente útil conocer cuando un objeto deriva de un constructor específico. Para esto, JavaScript tiene un operador binario llamado `instanceof`.

```
console.log(new DCeldaTexto("A") instanceof DCeldaTexto);
// → true
console.log(new DCeldaTexto("A") instanceof CeldaTexto);
// → true
console.log(new CeldaTexto("A") instanceof DCeldaTexto);
// → false
console.log([1] instanceof Array);
// → true
```

El operador verá a través de los tipos heredados. Una `DCeldaTexto` es una instancia de `CeldaTexto` porque `DCeldaTexto.prototype` deriva de `CeldaTexto.prototype`. El operador puede ser aplicado a constructores estándar como `Array`. Aunque casi todos los objetos son una instancia de `Object`.

Resumen

Entonces los objetos son más complicados de lo que inicialmente he mostrado. Tienen prototipos, que son otros objetos, y actuarán como si tuviesen las propiedades que no tienen, si las tienen sus prototipos. Los objetos simples tienen `Object.prototype` como su prototipo.

Los constructores, que son funciones cuyos nombres normalmente empiezan con una mayúscula, pueden ser usados con el operador `new` para crear nuevos objetos. Los nuevos prototipos de los objetos se encontrarán en la propiedad `prototype` de la función

constructora. Puedes hacer buen uso de esto poniendo las propiedades que comparten todos los valores de un tipo dado en su prototipo. El operador `instanceOf` puede, dado un objeto y un constructor, decirte cuando ese objeto es una instancia de ese constructor.

Algo útil para hacer con objetos es especificar una interfaz para ellos y decir a todos los que van a comunicarse con el objeto que lo hagan solo a través de la interfaz. El resto de detalles que maquillan tu objeto son ahora *encapsulados*, ocultados tras la interfaz.

Ahora que estamos hablando en términos de interfaces, ¿quien dice que solo un tipo de objeto puede implementarse en esa interfaz? Tener diferentes objetos expuestos a la misma interfaz y después escribir código que funcione en cualquier objeto, es la interfaz llamada *polimórfica*. Esto es muy útil.

Cuando implementamos múltiples tipos que difieren solo en algunos detalles, esto puede ayudar a simplificar haciendo el que el prototipo del nuevo tipo de objeto derive del prototipo del viejo y tener un nuevo constructor que puede llamar al antiguo. Esto te da un tipo de objeto similar al viejo pero puedes añadir y sobre escribir propiedades que veas que encajan.

Ejercicios

Un tipo vector

Escribe un *constructor* `Vector` que represente un vector en un espacio de dos dimensiones. Este toma `x` e `y` como parámetros (números), que se deben guardar como propiedades del mismo nombre.

Añade al prototipo `Vector` dos métodos, `mas` y `menos`, que toman otro vector como parámetro y devuelven un nuevo vector con el resultado de la suma o resta de los dos vectores (el vector almacenado en `this` y el parámetro) con sus valores *x* e *y*.

Añade una propiedad *getter* `longitud` al prototipo que calcule la longitud del vector-esto es, la distancia del punto (*x*, *y*) desde el origen (0,0).

```
// Your code here.

console.log(new Vector(1, 2).mas(new Vector(2, 3)));
// → Vector{x: 3, y: 5}
console.log(new Vector(1, 2).menos(new Vector(2, 3)));
// → Vector{x: -1, y: -1}
console.log(new Vector(3, 4).longitud);
// → 5
```

!!pista!!

Tu solución se puede acercar mucho al patrón del constructor `Conejo` de este capítulo.

Añadir una propiedad getter al constructor se puede hacer con la función `Object.defineProperty`. Para calcular la distancia de (0,0) a (x,y), puedes usar el teorema de Pitágoras, que dice que el cuadrado de la distancia que buscamos es igual a la suma de los cuadrados de la coordenada-x y la coordenada-y. Por tanto, (!html √(x^2^ + y^2^pass:[)]!)(!tex pass:[$\sqrt{x^2 + y^2}$]!) es el número que buscas, y `Math.sqrt` es la forma en la que calculas una raíz cuadrada en JavaScript.

Otra celda

Implementa un tipo de celda llamado `CeldaEstirar(contenido, anchura, altura)` que se ajuste a la interfaz tabla celda] descrita previamente en el capítulo. Esta debe contener otra celda (como hace `CeldaSurayada`) y asegurar que la celda resultante tiene al menos la `anchura` y `altura` dadas, incluso si el contenido de la celda pueda ser naturalmente menor.

```
// Your code here.

var sc = new CeldaEstirar(new CeldaTexto("abc"), 1, 2);
console.log(sc.minAnchura());
// → 3
console.log(sc.minAltura());
// → 2
console.log(sc.dibujar(3, 2));
// → ["abc", "   "]
```

!!pista!!

Tienes que guardar los tres argumentos de la instancia en el constructor. Los métodos `minAnchura` y `minAltura` deben llamarse a través de los correspondientes métodos en el `contenido` de la celda pero asegurar que no se retorna un número menor que el tamaño dado (probablemente usando `Math.max`).

No olvides añadir un método `draw` que simplemente redireccione la llamada al contenido de la celda.

Interface secuencia

Diseña una *interfaz* que resuma la *iteración* sobre una *colección* de valores. El objeto que provee a esta interfaz representa una secuencia. La interfaz debe mostrar como se hace esto posible usando un objeto para iterar sobre la secuencia, mirando los valores que tienen el elemento y con forma de detectar cuando se ha llegado al final de la secuencia.

Cuando hayas especificado tu interfaz, intenta escribir una función `mostrarCinco` que tome el objeto secuencia y llame a `console.log` en sus primeros cinco elementos-o menos, si la secuencia tiene menos de cinco elementos.

Después implementa un objeto del tipo `ArraySec` que contenga un array y permita la iteración sobre el array usando la interfaz que has diseñado. Implementa otro tipo de objeto `RangoSec` que itere sobre un rango de enteros (tomando los argumentos `desde` y `hasta` en su constructor) en su lugar.

```
// Your code here.

mostrarCinco(new ArraySeq([1, 2]));
// → 1
// → 2
mostrarCinco(new RangeSeq(100, 1000));
// → 100
// → 101
// → 102
// → 103
// → 104
```

!!pista!!

Una forma de resolver esto es dar a los objetos secuencia un *estado*, implicando que sus propiedades son cambiadas mientras se esta utilizando. Puedes guardar un contador que indique como de lejos ha llegado la secuencia.

Tu *interfaz* necesitará contar con al menos una forma de obtener el siguiente elemento y calcular si la iteración ha llegado al final de secuencia o no. Es tentador incluir esto en un método, `siguiente` , que retorne `null` o `undefined` cuando la secuencia haya llegado a su fin. Pero ahora tienes un problema cuando una secuencia actualmente contiene `null` . A sí que un método separado (o una propiedad getter) para encontrar cuando se ha llegado al final es probablemente preferible.

Otra solución es evitar cambiar el estado en el objeto. Puedes tener un método para devolver el elemento actual (sin avanzar ningún contador) y otro para obtener una nueva secuencia que represente los elementos restantes después del actual (o un valor especial cuando se llega al final de la secuencia). Esto es bastante elegante-un valor secuencia que “dependa de si mismo” incluso si después es usado y por tanto pueda ser compartido con otro código sin preocuparse de que puede pasar. Esto es, desafortunadamente, incluso algo ineficiente en un leguaje como JavaScript porque implica la creación de muchos objetos durante la iteración.

Proyecto: Vida Electronica

> [...] La pregunta de si las máquinas pueden pensar [...] es tan relevante como la pregunta de si los submarinos pueden nadar. Edsger Dijkstra, The Threats to Computing Science

En los capítulos de proyecto, dejaré de abrumarte con teoría nueva por un breve momento, y en lugar de eso trabajaremos a través de un programa juntos. La teoría es indispensable cuando aprendemos a programar, pero debería ser acompañada de lecturas y la comprensión de programas no triviales.

Nuestro proyecto en este capítulo es construir un ecosistema virtual, un mundo pequeño poblado con bichos que se mueven alrededor y luchan por sobrevivir.

Definición

Para hacer esta tarea manejable, nosotros simplificaremos radicalmente el concepto de *mundo*. Es decir un mundo será una *cuadricula* de dos dimensiones donde cada entidad ocupa un cuadro completo de ella. En cada *turno*, todos los bichos tienen oportunidad de tomar alguna acción.

Por lo tanto, cortamos ambos tiempo y espacio en dos unidades con un tamaño fijo: cuadros para espacio y turnos para tiempo. Por supuesto, esto es una burda e imprecisa *aproximación*. Pero nuestra simulación pretende ser entretenida, no precisa, así que podemos cortar libremente las esquinas.

Podemos definir un mundo con un *plan*, una matriz de cadenas que establece la cuadrícula del mundo usando un carácter por cuadro.

```
var plan = ["############################",
            "#      #    #      o      ##",
            "#                          #",
            "#          #####           #",
            "##         #   #    ##     #",
            "###           ##     #     #",
            "#           ###      #     #",
            "#   ####                   #",
            "#   ##       o             #",
            "# o  #         o       ### #",
            "#    #                     #",
            "############################"];
```

El carácter "#" en este programa representa *paredes* y rocas, y el carácter "o" representa bichos. Los espacios, como posiblemente habrás adivinado, son espacios vacíos.

Una matriz unidimensional puede ser usada para crear un objeto *mundo*. Tal objeto mantiene seguimiento del tamaño y contenido del mundo tiene un método de "toString", que convierte al mundo nuevamente en una cadena imprimible (parecida al programa en el que se basó) de manera que podamos ver qué es lo que está pasando dentro. El objeto mundo también tiene un método de `vuelta`, el cual permite a todos los bichos en él tomar un turno y actualizar el mundo a reflejo de sus acciones.

Representando el espacio.

La *cuadrícula* que modela el mundo tiene un ancho y altura fija. Los cuadros son identificados por sus coordenadas "X" y "Y". Usamos un tipo sencillo, `Vector` (como los vistos en los ejercicios del capítulo anterior), para representar estas coordenadas en pares.

```
function Vector(x, y) {
  this.x = x;
  this.y = y;
}
Vector.prototype.plus = function(other) {
  return new Vector(this.x + other.x, this.y + other.y);
};
```

A continuacion, necesitamos un tipo de objeto que modele por si mismo la cuadricula. Una cuadricula es parte de un mundo, pero nosotros estamos haciendo un objeto separado (la cuál sera una propiedad de un objeto del *mundo*) para mantener el objeto mundo simple. El mundo debe ocuparse de las cosas relacionadas con mundo, y la cuadricula debe ocuparse de las cosas relacionadas con la cuadricula.

Para almacenar una cuadricula de valores, tenemos varias opciones. Podemos utilizar una matriz de matrices de fila y utilizar dos propiedades de acceso para llegar a una cruadricula específica, como esto:

```
var grid = [["top left",    "top middle",    "top right"],
            ["bottom left", "bottom middle", "bottom right"]];
console.log(grid[1][2]);
// → bottom right
```

O podemos utilizar una sola matriz, con el tamaño de ancho x alto, y decidir que el elemento en (*x*,*y*) se encuentra en la posición *x* + (*y* x ancho) de la matriz.

```
var grid = ["top left",    "top middle",    "top right",
            "bottom left", "bottom middle", "bottom right"];
console.log(grid[2 + (1 * 3)]);
// → bottom right
```

Dado que el acceso real a esta matriz será envuelto en métodos en el objeto de tipo cuadricula, no le importa al código externo cual enfoque tomamos. Elegí la segunda representación, ya que hace que sea mucho más fácil crear la matriz. Al llamar al constructor de `Array` con un solo número como argumento, se crea una nueva matriz vacía de la longitud dada.

Este código define el objeto de cuadrícula, con algunos métodos básicos:

```
function Grid(width, height) {
  this.space = new Array(width * height);
  this.width = width;
  this.height = height;
}
Grid.prototype.isInside = function(vector) {
  return vector.x >= 0 && vector.x < this.width &&
         vector.y >= 0 && vector.y < this.height;
};
Grid.prototype.get = function(vector) {
  return this.space[vector.x + this.width * vector.y];
};
Grid.prototype.set = function(vector, value) {
  this.space[vector.x + this.width * vector.y] = value;
};
```

Y aquí es una prueba trivial:

```
var grid = new Grid(5, 5);
console.log(grid.get(new Vector(1, 1)));
// → undefined
grid.set(new Vector(1, 1), "X");
console.log(grid.get(new Vector(1, 1)));
// → X
```

Una interfaz de programación de bichos

Before we can start on the `World` ((constructor)), we must get more specific about the ((critter)) objects that will be living inside it. I mentioned that the world will ask the critters what actions they want to take. This works as follows: each critter object has an `act`

((method)) that, when called, returns an *action*. An action is an object with a `type` property, which names the type of action the critter wants to take, for example `"move"`. The action may also contain extra information, such as the direction the critter wants to move in.

Critters are terribly myopic and can see only the squares directly around them on the grid. But even this limited vision can be useful when deciding which action to take. When the `act` method is called, it is given a *view* object that allows the critter to inspect its surroundings. We name the eight surrounding squares by their ((compass direction))s: `"n"` for north, `"ne"` for northeast, and so on. Here's the object we will use to map from direction names to coordinate offsets:

```
var directions = {
  "n":  new Vector( 0, -1),
  "ne": new Vector( 1, -1),
  "e":  new Vector( 1,  0),
  "se": new Vector( 1,  1),
  "s":  new Vector( 0,  1),
  "sw": new Vector(-1,  1),
  "w":  new Vector(-1,  0),
  "nw": new Vector(-1, -1)
};
```

The view object has a method `look`, which takes a direction and returns a character, for example `"#"` when there is a wall in that direction, or `" "` (space) when there is nothing there. The object also provides the convenient methods `find` and `findAll`. Both take a map character as an argument. The first returns a direction in which the character can be found next to the critter or returns `null` if no such direction exists. The second returns an array containing all directions with that character. For example, a creature sitting left (west) of a wall will get `["ne", "e", "se"]` when calling `findAll` on its view object with the `"#"` character as argument.

sHere is a simple, stupid critter that just follows its nose until it hits an obstacle and then bounces off in a random open direction:

```
function randomElement(array) {
  return array[Math.floor(Math.random() * array.length)];
}

var directionNames = "n ne e se s sw w nw".split(" ");

function BouncingCritter() {
  this.direction = randomElement(directionNames);
};

BouncingCritter.prototype.act = function(view) {
  if (view.look(this.direction) != " ")
    this.direction = view.find(" ") || "s";
  return {type: "move", direction: this.direction};
};
```

The `randomElement` helper function simply picks a random element from an array, using `Math.random` plus some arithmetic to get a random index. We'll use this again later because randomness can be useful in ((simulation))s.

To pick a random direction, the `BouncingCritter` constructor calls `randomElement` on an array of direction names. We could also have used `Object.keys` to get this array from the `directions` object we defined link:07_elife.html#directions[earlier], but that provides no guarantees about the order in which the properties are listed. In most situations, modern JavaScript engines will return properties in the order they were defined, but they are not required to.

The “++|| "s"++” in the `act` method is there to prevent `this.direction` from getting the value `null` if the critter is somehow trapped with no empty space around it (for example when crowded into a corner by other critters).

== The world object ==

Now we can start on the `World` object type. The ((constructor)) takes a plan (the array of strings representing the world's grid, described link:07*elife.html#grid[earlier]) and a* ((legend))_ as arguments. A legend is an object that tells us what each character in the map means. It contains a constructor for every character—except for the space character, which always refers to `null`, the value we'll use to represent empty space.

```
function elementFromChar(legend, ch) {
  if (ch == " ")
    return null;
  var element = new legend[ch]();
  element.originChar = ch;
  return element;
}

function World(map, legend) {
  var grid = new Grid(map[0].length, map.length);
  this.grid = grid;
  this.legend = legend;

  map.forEach(function(line, y) {
    for (var x = 0; x < line.length; x++)
      grid.set(new Vector(x, y),
               elementFromChar(legend, line[x]));
  });
}
```

In `elementFromChar`, first we create an instance of the right type by looking up the character's constructor and applying `new` to it. Then we add an `originChar` ((property)) to it to make it easy to find out what character the element was originally created from.

We need this `originChar` property when implementing the world's `toString` method. This method builds up a maplike string from the world's current state by performing a two-dimensional loop over the squares on the grid.

```
function charFromElement(element) {
  if (element == null)
    return " ";
  else
    return element.originChar;
}

World.prototype.toString = function() {
  var output = "";
  for (var y = 0; y < this.grid.height; y++) {
    for (var x = 0; x < this.grid.width; x++) {
      var element = this.grid.get(new Vector(x, y));
      output += charFromElement(element);
    }
    output += "\n";
  }
  return output;
};
```

A ((wall)) is a simple object—it is used only for taking up space and has no `act` method.

```
function Wall() {}
```

When we try the `World` object by creating an instance based on the plan from link:07_elife.html#plan[earlier in the chapter] and then calling `toString` on it, we get a string very similar to the plan we put in.

// include_code strip_log // test: trim

```
var world = new World(plan, {"#": Wall,
                             "o": BouncingCritter});
console.log(world.toString());
// → ############################
//   #      #    #      o      ##
//   #                          #
//   #          #####           #
//   ##         #   #    ##     #
//   ###           ##     #     #
//   #           ###      #     #
//   #   ####                   #
//   #   ##       o             #
//   # o  #         o       ### #
//   #    #                     #
//   ############################
```

== this and its scope ==

The `World` ((constructor)) contains a call to `forEach`. One interesting thing to note is that inside the function passed to `forEach`, we are no longer directly in the function scope of the constructor. Each function call gets its own `this` binding, so the `this` in the inner function does *not* refer to the newly constructed object that the outer `this` refers to. In fact, when a function isn't called as a method, `this` will refer to the global object.

This means that we can't write `this.grid` to access the grid from inside the ((loop)). Instead, the outer function creates a normal local variable, `grid`, through which the inner function gets access to the grid.

This is a bit of a design blunder in JavaScript. Fortunately, the next version of the language provides a solution for this problem. Meanwhile, there are workarounds. A common pattern is to say `var self = this` and from then on refer to `self`, which is a normal variable and thus visible to inner functions.

(((bind method)))(((this)))Another solution is to use the `bind` method, which allows us to provide an explicit `this` object to bind to.

```
var test = {
  prop: 10,
  addPropTo: function(array) {
    return array.map(function(elt) {
      return this.prop + elt;
    }.bind(this));
  }
};
console.log(test.addPropTo([5]));
// → [15]
```

The function passed to `map` is the result of the `bind` call and thus has its `this` bound to the first argument given to ++bind++—the outer function's `this` value (which holds the `test` object).

Most ((standard)) higher-order methods on arrays, such as `forEach` and `map`, take an optional second argument that can also be used to provide a `this` for the calls to the iteration function. So you could express the previous example in a slightly simpler way.

```
var test = {
  prop: 10,
  addPropTo: function(array) {
    return array.map(function(elt) {
      return this.prop + elt;
    }, this); // ← no bind
  }
};
console.log(test.addPropTo([5]));
// → [15]
```

This works only for higher-order functions that support such a *context* parameter. When they don't, you'll need to use one of the other approaches.

In our own higher-order functions, we can support such a context parameter by using the `call` method to call the function given as an argument. For example, here is a `forEach` method for our `Grid` type, which calls a given function for each element in the grid that isn't null or undefined:

```
Grid.prototype.forEach = function(f, context) {
  for (var y = 0; y < this.height; y++) {
    for (var x = 0; x < this.width; x++) {
      var value = this.space[x + y * this.width];
      if (value != null)
        f.call(context, value, new Vector(x, y));
    }
  }
};
```

== Animating life ==

The next step is to write a `turn` method for the world object that gives the ((critter))s a chance to act. It will go over the grid using the `forEach` method we just defined, looking for objects with an `act` method. When it finds one, `turn` calls that method to get an action object and carries out the action when it is valid. For now, only `"move"` actions are understood.

(((grid)))There is one potential problem with this approach. Can you spot it? If we let critters move as we come across them, they may move to a square that we haven't looked at yet, and we'll allow them to move *again* when we reach that square. Thus, we have to keep an array of critters that have already had their turn and ignore them when we see them again.

```
World.prototype.turn = function() {
  var acted = [];
  this.grid.forEach(function(critter, vector) {
    if (critter.act && acted.indexOf(critter) == -1) {
      acted.push(critter);
      this.letAct(critter, vector);
    }
  }, this);
};
```

(((this)))We use the second parameter to the grid's `forEach` method to be able to access the correct `this` inside the inner function. The `letAct` method contains the actual logic that allows the critters to move.

```
World.prototype.letAct = function(critter, vector) {
  var action = critter.act(new View(this, vector));
  if (action && action.type == "move") {
    var dest = this.checkDestination(action, vector);
    if (dest && this.grid.get(dest) == null) {
      this.grid.set(vector, null);
      this.grid.set(dest, critter);
    }
  }
};

World.prototype.checkDestination = function(action, vector) {
  if (directions.hasOwnProperty(action.direction)) {
    var dest = vector.plus(directions[action.direction]);
    if (this.grid.isInside(dest))
      return dest;
  }
};
```

First, we simply ask the critter to act, passing it a view object that knows about the world and the critter's current position in that world (we'll define `View` in a link:07_elife.html#view[moment]). The `act` method returns an action of some kind.

If the action's `type` is not `"move"`, it is ignored. If it *is* `"move"`, if it has a `direction` property that refers to a valid direction, *and* if the square in that direction is empty (null), we set the square where the critter used to be to hold null and store the critter in the destination square.

Note that `letAct` takes care to ignore nonsense ((input))—it doesn't assume that the action's `direction` property is valid or that the `type` property makes sense. This kind of *defensive* programming makes sense in some situations. The main reason for doing it is to validate inputs coming from sources you don't control (such as user or file input), but it can also be useful to isolate subsystems from each other. In this case, the intention is that the critters themselves can be programmed sloppily—they don't have to verify if their intended actions make sense. They can just request an action, and the world will figure out whether to allow it.

These two methods are not part of the external interface of a `World` object. They are an internal detail. Some languages provide ways to explicitly declare certain methods and properties *private* and signal an error when you try to use them from outside the object. JavaScript does not, so you will have to rely on some other form of communication to describe what is part of an object's interface. Sometimes it can help to use a naming scheme to distinguish between external and internal properties, for example by prefixing all internal ones with an underscore character (_). This will make accidental uses of properties that are not part of an object's interface easier to spot.

The one missing part, the `View` type, looks like this:

```
function View(world, vector) {
  this.world = world;
  this.vector = vector;
}
View.prototype.look = function(dir) {
  var target = this.vector.plus(directions[dir]);
  if (this.world.grid.isInside(target))
    return charFromElement(this.world.grid.get(target));
  else
    return "#";
};
View.prototype.findAll = function(ch) {
  var found = [];
  for (var dir in directions)
    if (this.look(dir) == ch)
      found.push(dir);
  return found;
};
View.prototype.find = function(ch) {
  var found = this.findAll(ch);
  if (found.length == 0) return null;
  return randomElement(found);
};
```

The `look` method figures out the coordinates that we are trying to look at and, if they are inside the ((grid)), finds the character corresponding to the element that sits there. For coordinates outside the grid, `look` simply pretends that there is a wall there so that if you define a world that isn't walled in, the critters still won't be tempted to try to walk off the edges.

== It moves ==

We instantiated a world object earlier. Now that we've added all the necessary methods, it should be possible to actually make the world move.

```
for (var i = 0; i < 5; i++) {
  world.turn();
  console.log(world.toString());
}
// → … five turns of moving critters
```

The first two maps that are displayed will look something like this (depending on the random direction the critters picked):

```
############################  ############################
#      #    #            ##  #      #    #            ##
#                  o      #  #                          #
#          #####          #  #          #####     o     #
##         #   #    ##     #  ##         #   #    ##     #
###           ##     #     #  ###           ##     #     #
#           ###      #     #  #           ###      #     #
#   ####                   #  #   ####                   #
#   ##                     #  #   ##                     #
#    #       o         ### #  #o   #                ### #
#o   #           o          #  #    #       o o          #
############################  ############################
```

They move! To get a more interactive view of these critters crawling around and bouncing off the walls, open this chapter in the online version of the book at http://eloquentjavascript.net[_eloquentjavascript.net_].

Simply printing out many copies of the map is a rather unpleasant way to observe a world, though. That's why the sandbox provides an `animateWorld` function that will run a world as an onscreen animation, moving three turns per second, until you hit the stop button.

```
animateWorld(world);
// → … life!
```

The implementation of `animateWorld` will remain a mystery for now, but after you've read the link:13_dom.html#dom[later chapters] of this book, which discuss JavaScript integration in web browsers, it won't look so magical anymore.

== More life forms ==

The dramatic highlight of our world, if you watch for a bit, is when two critters bounce off each other. Can you think of another interesting form of ((behavior))?

The one I came up with is a ((critter)) that moves along walls. Conceptually, the critter keeps its left hand (paw, tentacle, whatever) to the wall and follows along. This turns out to be not entirely trivial to implement.

We need to be able to “compute” with ((compass direction))s. Since directions are modeled by a set of strings, we need to define our own operation (`dirPlus`) to calculate relative directions. So `dirPlus("n", 1)` means one 45-degree turn clockwise from north, giving `"ne"` . Similarly, `dirPlus("s", -2)` means 90 degrees counterclockwise from south, which is east.

```
function dirPlus(dir, n) {
  var index = directionNames.indexOf(dir);
  return directionNames[(index + n + 8) % 8];
}

function WallFollower() {
  this.dir = "s";
}

WallFollower.prototype.act = function(view) {
  var start = this.dir;
  if (view.look(dirPlus(this.dir, -3)) != " ")
    start = this.dir = dirPlus(this.dir, -2);
  while (view.look(this.dir) != " ") {
    this.dir = dirPlus(this.dir, 1);
    if (this.dir == start) break;
  }
  return {type: "move", direction: this.dir};
};
```

The `act` method only has to “scan” the critter's surroundings, starting from its left side and going clockwise until it finds an empty square. It then moves in the direction of that empty square.

What complicates things is that a critter may end up in the middle of empty space, either as its start position or as a result of walking around another critter. If we apply the approach I just described in empty space, the poor critter will just keep on turning left at every step, running in circles.

So there is an extra check (the `if` statement) to start scanning to the left only if it looks like the critter has just passed some kind of ((obstacle))—that is, if the space behind and to the left of the critter is not empty. Otherwise, the critter starts scanning directly ahead, so that it'll walk straight when in empty space.

And finally, there's a test comparing `this.dir` to `start` after every pass through the loop to make sure that the loop won't run forever when the critter is walled in or crowded in by other critters and can't find an empty square.

This small world demonstrates the wall-following creatures:

```
animateWorld(new World(
  ["#############",
   "#     #    #",
   "#   ~    ~ #",
   "#  ##      #",
   "#  ##  o####",
   "#          #",
   "#############"],
  {"#": Wall,
   "~": WallFollower,
   "o": BouncingCritter}
));
```

== A more lifelike simulation ==

To make life in our world more interesting, we will add the concepts of ((food)) and ((reproduction)). Each living thing in the world gets a new property, `energy`, which is reduced by performing actions and increased by eating things. When the critter has enough ((energy)), it can reproduce, generating a new critter of the same kind. To keep things simple, the critters in our world reproduce asexually, all by themselves.

If critters only move around and eat one another, the world will soon succumb to the law of increasing entropy, run out of energy, and become a lifeless wasteland. To prevent this from happening (too quickly, at least), we add ((plant))s to the world. Plants do not move. They just use ((photosynthesis)) to grow (that is, increase their energy) and reproduce.

To make this work, we'll need a world with a different `letAct` method. We could just replace the method of the `World` prototype, but I've become very attached to our simulation with the wall-following critters and would hate to break that old world.

One solution is to use ((inheritance)). We create a new ((constructor)), `LifelikeWorld`, whose prototype is based on the `World` prototype but which overrides the `letAct` method. The new `letAct` method delegates the work of actually performing an action to various functions stored in the `actionTypes` object.

```
function LifelikeWorld(map, legend) {
  World.call(this, map, legend);
}
LifelikeWorld.prototype = Object.create(World.prototype);

var actionTypes = Object.create(null);

LifelikeWorld.prototype.letAct = function(critter, vector) {
  var action = critter.act(new View(this, vector));
  var handled = action &&
    action.type in actionTypes &&
    actionTypes[action.type].call(this, critter,
                                  vector, action);
  if (!handled) {
    critter.energy -= 0.2;
    if (critter.energy <= 0)
      this.grid.set(vector, null);
  }
};
```

The new `letAct` method first checks whether an action was returned at all, then whether a handler function for this type of action exists, and finally whether that handler returned true, indicating that it successfully handled the action. Note the use of `call` to give the handler access to the world, through its `this` binding.

If the action didn't work for whatever reason, the default action is for the creature to simply wait. It loses one-fifth point of ((energy)), and if its energy level drops to zero or below, the creature dies and is removed from the grid.

== Action handlers ==

The simplest action a creature can perform is `"grow"`, used by ((plant))s. When an action object like `{type: "grow"}` is returned, the following handler method will be called:

```
actionTypes.grow = function(critter) {
  critter.energy += 0.5;
  return true;
};
```

Growing always succeeds and adds half a point to the plant's ((energy)) level.

Moving is more involved.

```
actionTypes.move = function(critter, vector, action) {
  var dest = this.checkDestination(action, vector);
  if (dest == null ||
      critter.energy <= 1 ||
      this.grid.get(dest) != null)
    return false;
  critter.energy -= 1;
  this.grid.set(vector, null);
  this.grid.set(dest, critter);
  return true;
};
```

This action first checks, using the `checkDestination` method defined link:07_elife.html#checkDestination[earlier], whether the action provides a valid destination. If not, or if the destination isn't empty, or if the critter lacks the required ((energy)), `move` returns false to indicate no action was taken. Otherwise, it moves the critter and subtracts the energy cost.

In addition to moving, critters can eat.

```
actionTypes.eat = function(critter, vector, action) {
  var dest = this.checkDestination(action, vector);
  var atDest = dest != null && this.grid.get(dest);
  if (!atDest || atDest.energy == null)
    return false;
  critter.energy += atDest.energy;
  this.grid.set(dest, null);
  return true;
};
```

Eating another ((critter)) also involves providing a valid destination square. This time, the destination must not be empty and must contain something with ((energy)), like a critter (but not a wall—walls are not edible). If so, the energy from the eaten is transferred to the eater, and the victim is removed from the grid.

And finally, we allow our critters to reproduce.

```
actionTypes.reproduce = function(critter, vector, action) {
  var baby = elementFromChar(this.legend,
                             critter.originChar);
  var dest = this.checkDestination(action, vector);
  if (dest == null ||
      critter.energy <= 2 * baby.energy ||
      this.grid.get(dest) != null)
    return false;
  critter.energy -= 2 * baby.energy;
  this.grid.set(dest, baby);
  return true;
};
```

Reproducing costs twice the ((energy)) level of the newborn critter. So we first create a (hypothetical) baby using `elementFromChar` on the critter's own origin character. Once we have a baby, we can find its energy level and test whether the parent has enough energy to successfully bring it into the world. We also require a valid (and empty) destination.

If everything is okay, the baby is put onto the grid (it is now no longer hypothetical), and the energy is spent.

== Populating the new world ==

We now have a ((framework)) to simulate these more lifelike creatures. We could put the critters from the old world into it, but they would just die since they don't have an ((energy)) property. So let's make new ones. First we'll write a ((plant)), which is a rather simple life-form.

```
function Plant() {
  this.energy = 3 + Math.random() * 4;
}
Plant.prototype.act = function(view) {
  if (this.energy > 15) {
    var space = view.find(" ");
    if (space)
      return {type: "reproduce", direction: space};
  }
  if (this.energy < 20)
    return {type: "grow"};
};
```

Plants start with an energy level between 3 and 7, randomized so that they don't all reproduce in the same turn. When a plant reaches 15 energy points and there is empty space nearby, it reproduces into that empty space. If a plant can't reproduce, it simply grows until it reaches energy level 20.

We now define a plant eater.

```
function PlantEater() {
  this.energy = 20;
}
PlantEater.prototype.act = function(view) {
  var space = view.find(" ");
  if (this.energy > 60 && space)
    return {type: "reproduce", direction: space};
  var plant = view.find("*");
  if (plant)
    return {type: "eat", direction: plant};
  if (space)
    return {type: "move", direction: space};
};
```

We'll use the `*` character for ((plant))s, so that's what this creature will look for when it searches for ((food)).

== Bringing it to life ==

And that gives us enough elements to try our new world. Imagine the following map as a grassy valley with a herd of ((herbivore))s in it, some boulders, and lush ((plant)) life everywhere.

```
var valley = new LifelikeWorld(
  ["############################",
   "#####                 ######",
   "##   ***                **##",
   "#   *##**         **  O  *##",
   "#    ***     O    ##**    *#",
   "#       O         ##***    #",
   "#                 ##**     #",
   "#   O       #*             #",
   "#*          #**       O    #",
   "#***        ##**    O    **#",
   "##****     ###***       *###",
   "############################"],
  {"#": Wall,
   "O": PlantEater,
   "*": Plant}
);
```

Let's see what happens if we run this. (!book These snapshots illustrate a typical run of this world.!)

```
animateWorld(valley);
```

```
############################  ############################
#####                ######  ##### **             ######
##   ***   0           *##  ##  ** *           0     ##
#   *##*          **     *##  #  **##                  ##
#    **           ##*     *#  #  **  0         ##0      #
#                 ##*      #  #   *0      * *   ##       #
#                 ##  0    #  #            ***  ##     0 #
#           #*      0      #  #**          #***          #
#*          #**  0         #  #**      0  #****         #
#*   0    0 ##*          **#  #***        ##***     0    #
##*        ###*          ###  ##**       ###**    0    ###
############################  ############################

############################  ############################
#####0 0             ######  #####  0             ######
##                      ##  ##                      ##
#    ##0                ##  #    ##           0      ##
#           0  0 *##      #  #                ##       #
#  0    0    0  **##    0  #  #                ##       #
#               **##     0 #  #              0 ## *     #
#           #   *** *      #  #           #  0          #
#           # 0*****  0    #  #        0  #   0         #
#           ##******       #  #           ##    0     0  #
##         ###******     ###  ##         ### 0         ###
############################  ############################

############################  ############################
#####                ######  #####                ######
##                      ##  ##                ** * ##
#    ##                ##  #    ##           ***** ##
#                ##      #  #                ##****  #
#                ##* *   #  #                ##***** #
#             0  ## *    #  #                ##****** #
#           #            #  #           #      ** **  #
#           #            #  #           #            #
#           ##           #  #           ##           #
##         ###          ###  ##         ###          ###
############################  ############################
```

Most of the time, the plants multiply and expand quite quickly, but then the abundance of ((food)) causes a population explosion of the ((herbivore))s, who proceed to wipe out all or nearly all of the ((plant))s, resulting in a mass starvation of the critters. Sometimes, the ((ecosystem)) recovers and another cycle starts. At other times, one of the species dies out completely. If it's the herbivores, the whole space will fill with plants. If it's the plants, the remaining critters starve, and the valley becomes a desolate wasteland. Ah, the cruelty of nature.

== Exercises ==

=== Artificial stupidity ===

Having the inhabitants of our world go extinct after a few minutes is kind of depressing. To deal with this, we could try to create a smarter plant eater.

There are several obvious problems with our herbivores. First, they are terribly greedy, stuffing themselves with every plant they see until they have wiped out the local plant life. Second, their randomized movement (recall that the `view.find` method returns a random direction when multiple directions match) causes them to stumble around ineffectively and starve if there don't happen to be any plants nearby. And finally, they breed very fast, which makes the cycles between abundance and famine quite intense.

Write a new critter type that tries to address one or more of these points and substitute it for the old `PlantEater` type in the valley world. See how it fares. Tweak it some more if necessary.

// test: no

```
// Your code here
function SmartPlantEater() {}

animateWorld(new LifelikeWorld(
  ["############################",
   "#####                 ######",
   "##   ***                **##",
   "#   *##**         **  O  *##",
   "#    ***     O    ##**    *#",
   "#       O         ##***    #",
   "#                 ##**     #",
   "#   O       #*             #",
   "#*          #**       O    #",
   "#***        ##**    O    **#",
   "##****     ###***       *###",
   "############################"],
  {"#": Wall,
   "O": SmartPlantEater,
   "*": Plant}
));
```

!!hint!!

The greediness problem can be attacked in several ways. The critters could stop eating when they reach a certain ((energy)) level. Or they could eat only every N turns (by keeping a counter of the turns since their last meal in a property on the creature object). Or, to make sure plants never go entirely extinct, the animals could refuse to eat a ((plant)) unless they see at least one other plant nearby (using the `findAll` method on the view). A combination of these, or some entirely different strategy, might also work.

Making the critters move more effectively could be done by stealing one of the movement strategies from the critters in our old, energyless world. Both the bouncing behavior and the wall-following behavior showed a much wider range of movement than completely random staggering.

Making creatures breed more slowly is trivial. Just increase the minimum energy level at which they reproduce. Of course, making the ecosystem more stable also makes it more boring. If you have a handful of fat, immobile critters forever munching on a sea of plants and never reproducing, that makes for a very stable ecosystem. But no one wants to watch that.

!!hint!!

=== Predators ===

Any serious ((ecosystem)) has a food chain longer than a single link. Write another ((critter)) that survives by eating the ((herbivore)) critter. You'll notice that ((stability)) is even harder to achieve now that there are cycles at multiple levels. Try to find a strategy to make the ecosystem run smoothly for at least a little while.

One thing that will help is to make the world bigger. This way, local population booms or busts are less likely to wipe out a species entirely, and there is space for the relatively large prey population needed to sustain a small predator population.

```
// Your code here
function Tiger() {}

animateWorld(new LifelikeWorld(
  ["############################################################",
   "#                 ####         ****              ###",
   "#   *  @  ##                 ########       OO    ##",
   "#   *    ##        O O                 ****       *#",
   "#       ##*                        ##########     *#",
   "#      ##***  *         ****                     **#",
   "#* **  #  *  ***      #########                  **#",
   "#* **  #      *               #   *              **#",
   "#     ##              #   O   #  ***          ######",
   "#*            @       #       #   *        O  #    #",
   "#*                    #  ######                ** #",
   "###          ****          ***                 ** #",
   "#       O                        @         O       #",
   "#   *     ##  ##  ##  ##               ###      *  #",
   "#   **         #              *       #####  O     #",
   "##  **  O   O  #  #    ***  ***        ###      ** #",
   "###               #   *****                    ****#",
   "############################################################"],
  {"#": Wall,
   "@": Tiger,
   "O": SmartPlantEater, // from previous exercise
   "*": Plant}
));
```

!!hint!!

Many of the same tricks that worked for the previous exercise also apply here. Making the predators big (lots of energy) and having them reproduce slowly is recommended. That'll make them less vulnerable to periods of starvation when the herbivores are scarce.

Beyond staying alive, keeping its ((food)) stock alive is a predator's main objective. Find some way to make predators hunt more aggressively when there are a lot of ((herbivore))s and hunt more slowly (or not at all) when prey is rare. Since plant eaters move around, the simple trick of eating one only when others are nearby is unlikely to work—that'll happen so rarely that your predator will starve. But you could keep track of observations in previous turns, in some ((data structure)) kept on the predator objects, and have it base its ((behavior)) on what it has seen recently.